Shorabh Dhandharia
A20361514

HW4 – Support Vector Machines

Problem Statement

In this assignment, I have implemented different algorithms for Support Vector Machines. I have used iris and a linearly non separable dataset. I have also used a dataset "pima-indians-diabetes" having missing features. For cross validation, I have used previous assignment's technique of 10-fold cross validation. I have used the Python's inbuilt function to compute and confusion matrix and accuracy.

Proposed Solution

1. Generated a small dataset of 2D with two classes which are linearly separable and similarly with non-linearly separable.
2. Implemented a Linear SVM with Hard Margins on separable dataset and observe the results on non-separable dataset. Plotted the dataset and marked the support vectors.
3. Minimized the primal objective function of SVM using partial derivatives and develop the expression of the dual that has to be maximized
4. Implemented a Linear SVM with Soft Margins on separable dataset and observe the results on non-separable dataset. Plotted the dataset and marked the support vectors.
5. Implemented a kernel-based SVM using polynomial and Gaussian kernel functions. Applied the algorithm on the datasets generated and tested on iris and non-separable datasets for performance.
6. Tested the algorithm for datasets with one class having substantially more examples.

Implementation Details

I have implemented a small definition which will create a dataset containing no. of samples provided for each of the classes. For linearly non-separable dataset I am modifying the deviation in same function which will make few examples of 2 classes collide with each other.

For external datasets I have used iris dataset and non-separable dataset downloaded from Stanford's website.

In the code, I have imported the "cvxopt" package of python to calculate the lagrange multipliers. The algorithm is same as in previous assignments,

- Creating or Reading the entire dataset into Z matrix
- Reshuffling the rows to perform cross validation correctly
- Split the matrix into X and Y
- Strip the data to get float values
- In case of external dataset, modify the labels to -1 and 1
- Apply Cross Validation
    - Divide the Data into Training and Test
    - Calculate the matrices and vectors required for qp solvers method in cvxopt package

- Call the solvers.qp method and provide all the required arguments in cvxopt matrix format
- Get the lagrange multipliers from it.
- Identify the support vectors and get their X, Y and lagrange multipliers.
- Calculate w and w0
- Classify the examples and compute different measures

The value of c in soft margin is 1 and in case of hard margin I have used a higher value such as 10000. The value of epsilon is same in both the cases as 0.005.

Results and Discussions

Generated Dataset

I have generated a dataset for both linearly separable and non-separable. As we can see after applying cross fold, in case of separable dataset the accuracy is 1.0 whereas in case of non-separable dataset it is 0.95. This observation can be found for both Hard and Soft Margin

Cross Validation:

With Cross Fold (Separable dataset) [[8 0] [0 12]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0) *Same for Python's inbuilt function*	With Cross Fold (Non Separable dataset) [[7 0] [1 12]] ('Accuracy: ', 0.95) ('Precision: ', 0.875) ('Recall: ', 1.0) ('F-Measure: ', 0.93) *Python's inbuilt function classified successfully*

Without Cross Validation:

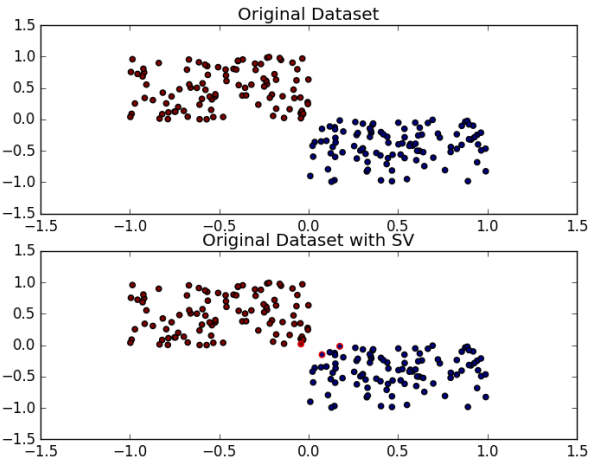
Without applying cross fold, the accuracy is still 1.0 for separable dataset, but the accuracy goes down to 0.88 for non-separable dataset. The same results are given by Python's inbuilt function for both Hard and Soft Margin.

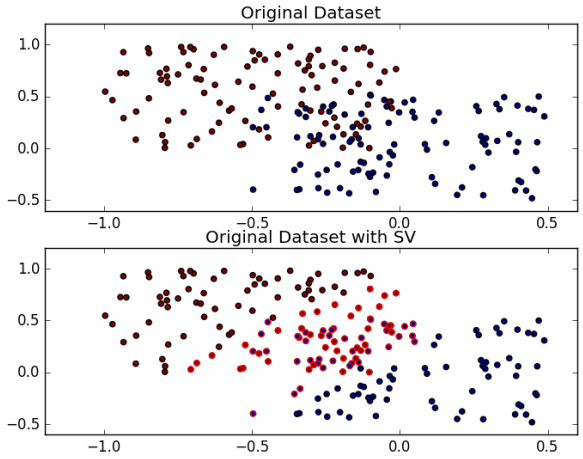
Without Cross Fold (Separable dataset) [[100 0] [0 100]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0) *Same for Python's inbuilt function*	Without Cross Fold (Non Separable dataset) Confusion Matrix: [[86 14] [10 90]] ('Accuracy: ', 0.88) ('Precision: ', 0.86538461538461542) ('Recall: ', 0.90000000000000002) ('F-Measure: ', 0.88235294117647067) *Python's inbuilt function*

	Confusion Matrix: [[83 17] [8 92]] ('Accuracy: ', 0.875) ('Precision: ', 0.84403669724770647) ('Recall: ', 0.92000000000000004) ('F-Measure: ', 0.88038277511961716)

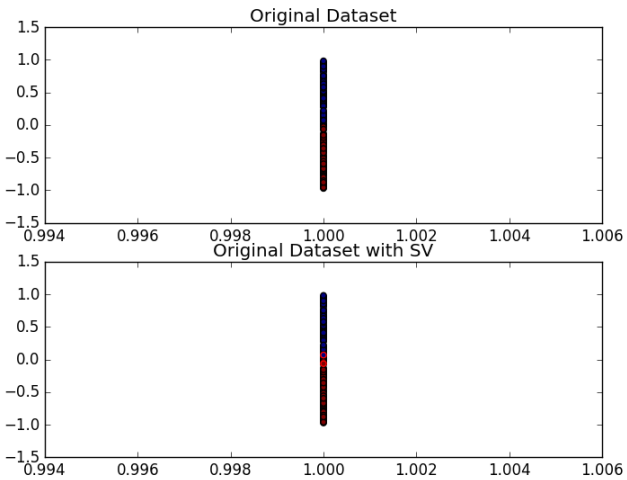
Hard Margin

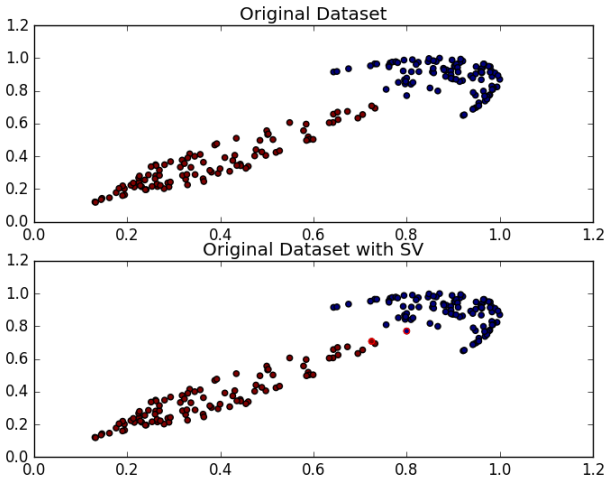
As explained above, below are the graphs for hard margin for separable and non-separable dataset**.** The support vectors can be identified in red colors.





The below graphs are for polynomial and radial kernels for separable dataset

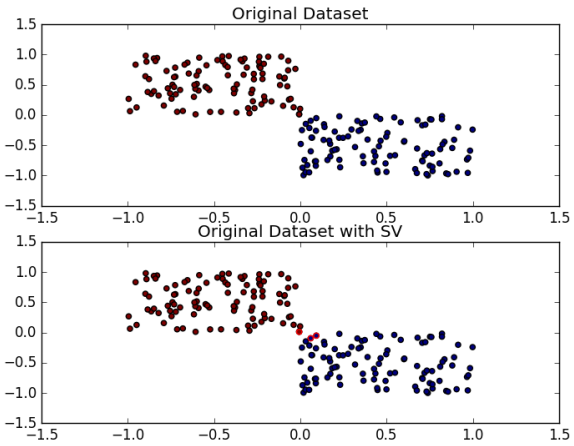




Soft Margin

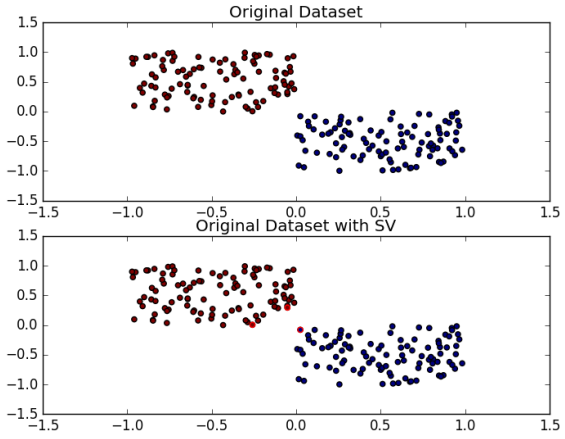
Below are the graphs for soft margin. The first one is with cross validation and separable dataset.

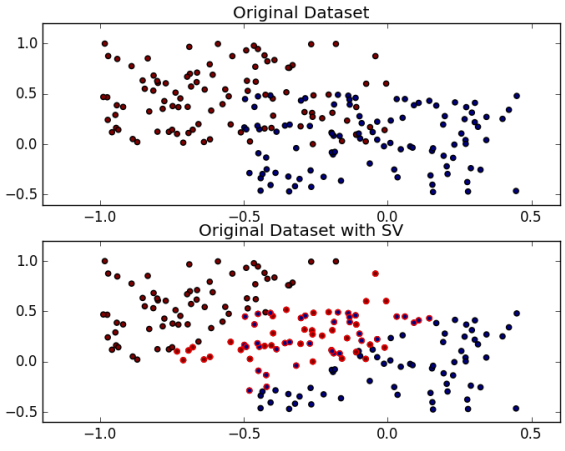
With Cross Validation



Without Cross Validation

Below are the graphs for separable and non-separable dataset without applying cross validation.

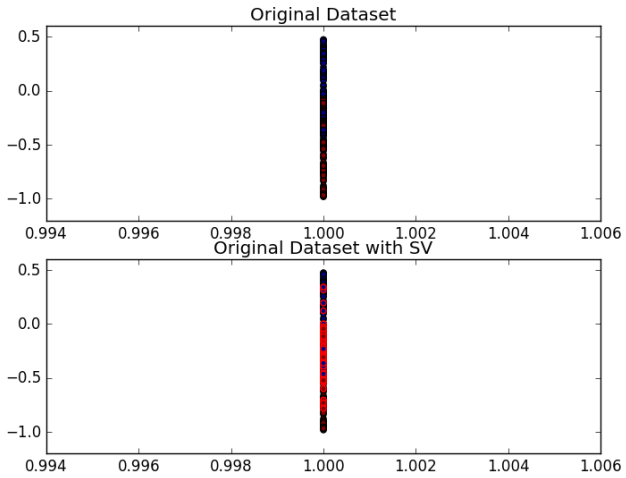


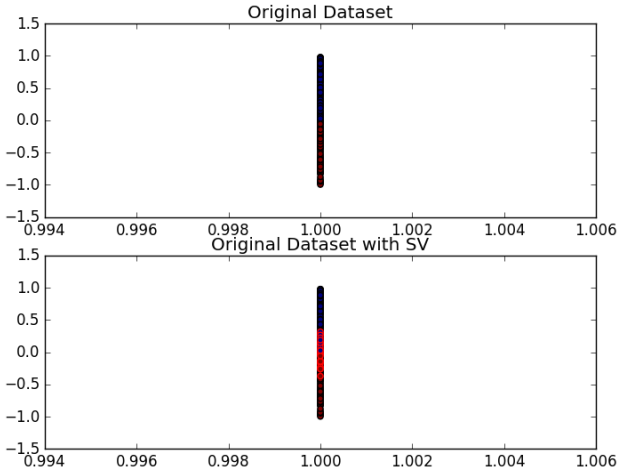


Polynomial Kernel with Linear separable and non-linearly separable

In this case, I have mapped the features to higher dimension of degree 3. The below table shows that, in case of separable dataset the accuracy remains the same as 1.0, but for the non-separable dataset it goes up to 0.93 from 0.88 in case of linear. The Python's inbuilt function gives accuracy slightly more than that. The graph below depicts the same. Left one is for Separable and right one is for non-separable. The support vectors can be identified in red colors.

Separable Dataset	Non-Separable Dataset
[[100 0] [0 100]] 1.0 ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0) *Same for Python's inbuilt function*	Confusion Matrix: [[91 9] [5 95]] ('Accuracy: ', 0.93000000000000005) ('Precision: ', 0.91346153846153844) ('Recall: ', 0.94999999999999996) ('F-Measure: ', 0.93137254901960775) ***Python's inbuilt function*** Confusion Matrix: [[95 5] [6 94]] ('Accuracy: ', 0.94499999999999995) ('Precision: ', 0.9494949494949495) ('Recall: ', 0.93999999999999995) ('F-Measure: ', 0.94472361809045236)

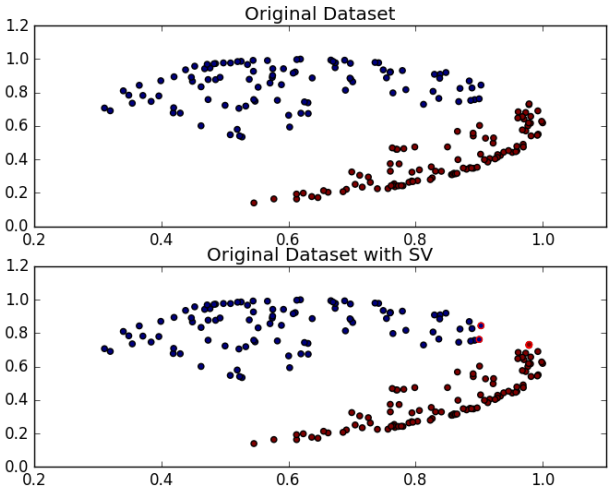


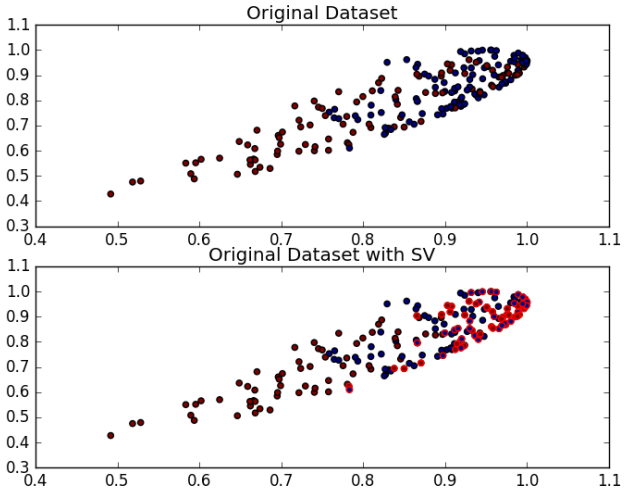


<u>Radial Kernel with Linear separable and non-linearly separable</u>

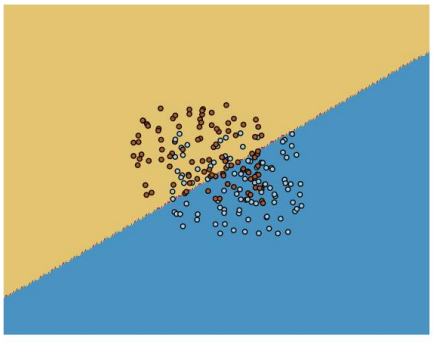
Here I have used radial kernel for both separable and non-separable datasets. Even in this case the separable dataset's accuracy remains 1 and non-separable dataset's accuracy is same as polynomial kernel. The results for Python's inbuilt function also remains the same as polynomial kernel. The graphs below depict the same. Left one is for Separable and right one is for non-separable. The support vectors can be identified in red colors.

Separable Dataset	Non-Separable Dataset
[[100 0] [0 100]] 1.0 ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0) *Same for Python's inbuilt function*	Confusion Matrix: [[91 9] [5 95]] ('Accuracy: ', 0.93000000000000005) ('Precision: ', 0.91346153846153844) ('Recall: ', 0.94999999999999996) ('F-Measure: ', 0.93137254901960775) ***Python's inbuilt function*** Confusion Matrix: [[95 5] [6 94]] ('Accuracy: ', 0.94499999999999995) ('Precision: ', 0.9494949494949495) ('Recall: ', 0.93999999999999995) ('F-Measure: ', 0.94472361809045236)





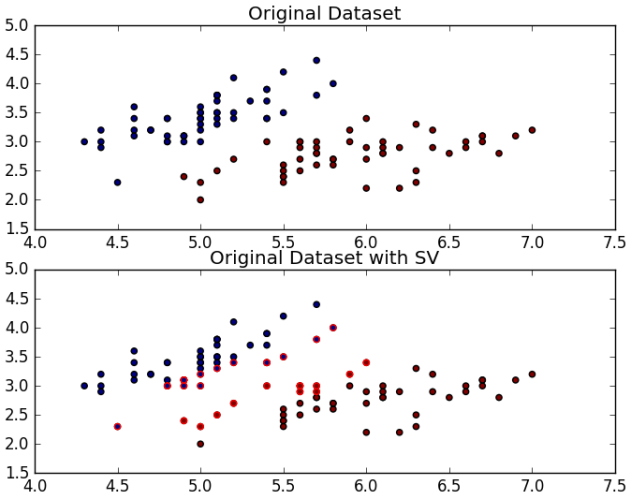
The below graph is a linear implementation of separable dataset with decision boundary. Its accuracy is same as that of linear separable dataset which is 1.0. The decision boundary graph for polynomial and radial kernels wasn't able to plot due to high number of feature vector.

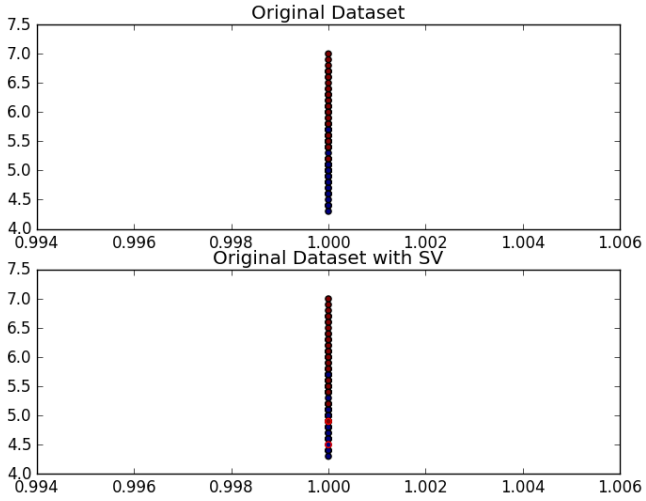


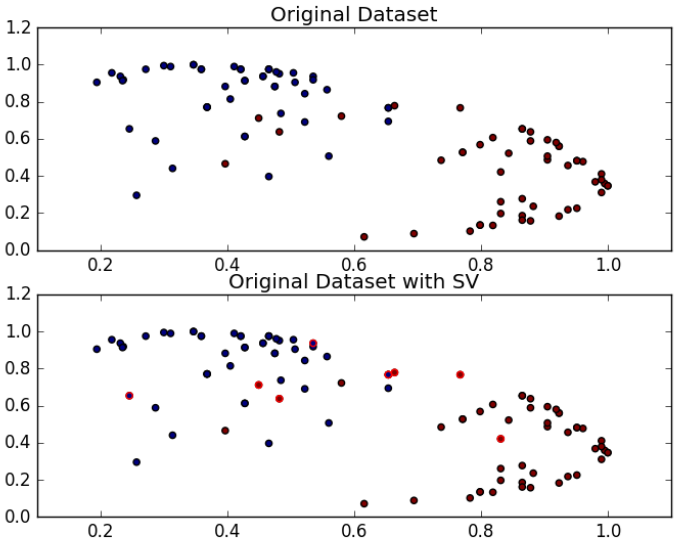
2. *Iris Dataset*

The same experiments were performed on iris dataset. Since it's a separable dataset the accuracy remains same for all the 3 kernels as mentioned in the below table. The below graphs are for linear, polynomial degree 3 and radial kernels. The support vectors can be identified in red colors.

Linear Confusion Matrix: [[50 0] [0 50]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0)	***Python's Inbuilt Function*** Confusion Matrix: [[50 0] [0 50]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0)
Same for Polynomial and RBF Kernel	



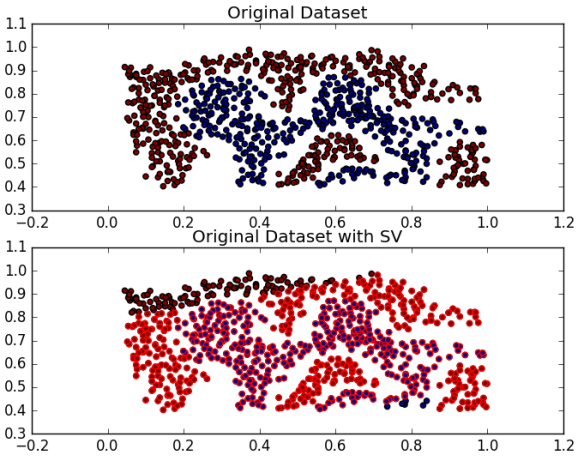


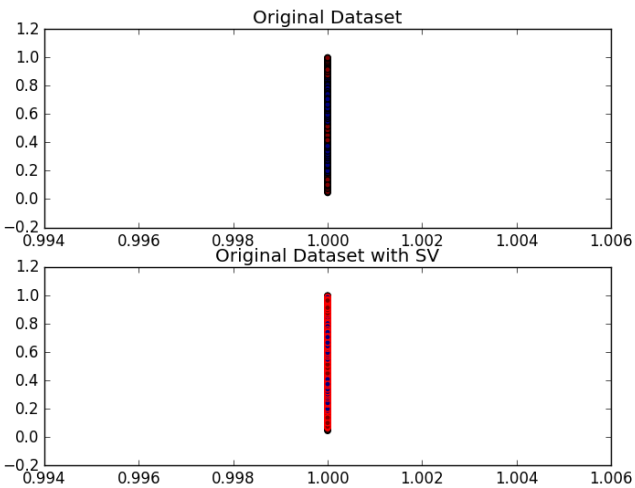


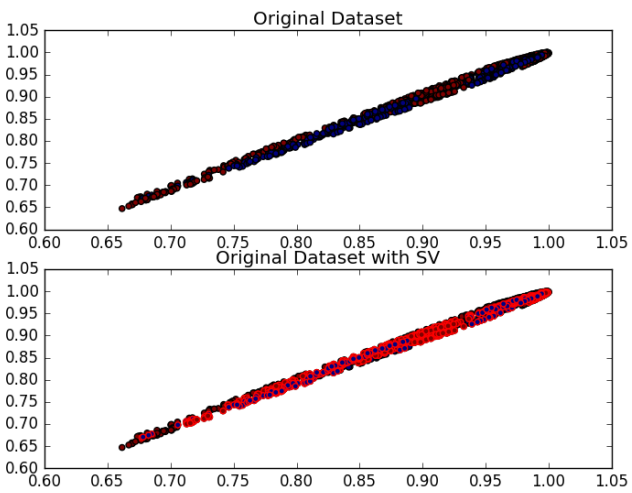
3. Stanford dataset for linearly non-separable.

This dataset is a non-separable dataset downloaded from the Stanford website. The same experiments were performed on this dataset also. Since it's a non-separable dataset the accuracy varies for all the 3 kernels as mentioned in the below table. In case of polynomial kernel, the Python's inbuilt function wasn't able to classify one of the classes. The below graphs are for linear, polynomial degree 3 and radial kernels. The support vectors can be identified in red colors.

Linear	Polynomial Kernel	RBF Kernel
Confusion Matrix:	Confusion Matrix:	Confusion Matrix:
[[187 196]	[[289 94]	[[293 90]
[167 313]]	[97 383]]	[93 387]]
('Accuracy: ', 0.58)	('Accuracy: ', 0.78)	('Accuracy: ', 0.79)
('Precision: ', 0.61)	('Precision: ', 0.80)	('Precision: ', 0.81)
('Recall: ', 0.65)	('Recall: ', 0.79)	('Recall: ', 0.81)
('F-Measure: ', 0.63)	('F-Measure: ', 0.80)	('F-Measure: ', 0.81)
Python's Inbuilt Function	***Python's Inbuilt Function***	***Python's Inbuilt Function***
Confusion Matrix:	Confusion Matrix:	Confusion Matrix:
[[175 208]	[[0 383]	[[382 1]
[164 316]]	[0 480]]	[4 476]]
('Accuracy: ', 0.57)	('Accuracy: ', 0.56)	('Accuracy: ', 0.99)
('Precision: ', 0.60)	('Precision: ', 0.56)	('Precision: ', 0.99)
('Recall: ', 0.66)	('Recall: ', 1.0)	('Recall: ', 0.99)
('F-Measure: ', 0.63)	('F-Measure: ', 0.71)	('F-Measure: ', 0.99)



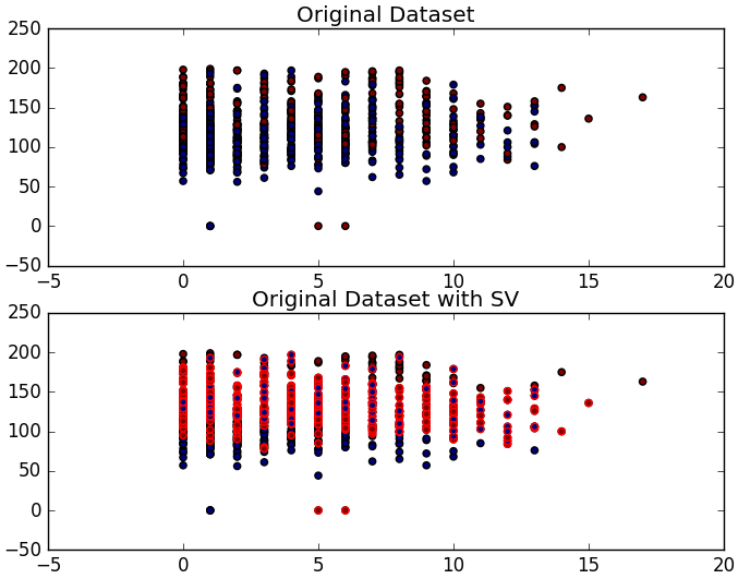




4. Pima Indians Diabetes Dataset (Missing Features)

This dataset is downloaded from the UCI website. It has 8 features containing missing values. I have chosen first two features for my experiment which contained around 100 missing values in 800 examples. The below table shows the results for my and Python's implementation which is almost the same. Both gives the accuracy of around 0.74. The graph depicts the same.

Confusion Matrix: [[413 87] [112 156]] ('Accuracy: ', 0.74088541666666663) ('Precision: ', 0.64197530864197527) ('Recall: ', 0.58208955223880599) ('F-Measure: ', 0.61056751467710368) ***Python's Inbuilt Function*** Confusion Matrix: [[441 59] [135 133]] ('Accuracy: ', 0.74739583333333337) ('Precision: ', 0.69270833333333337) ('Recall: ', 0.4962686567164179) ('F-Measure: ', 0.57826086956521738)

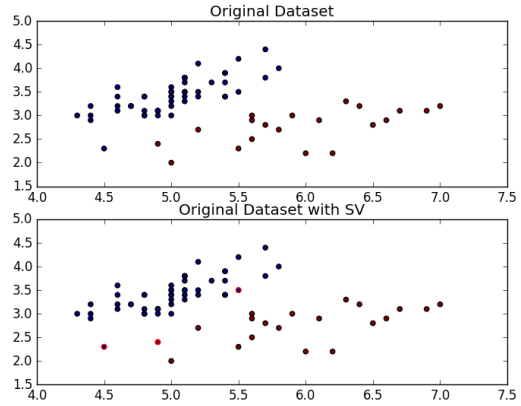


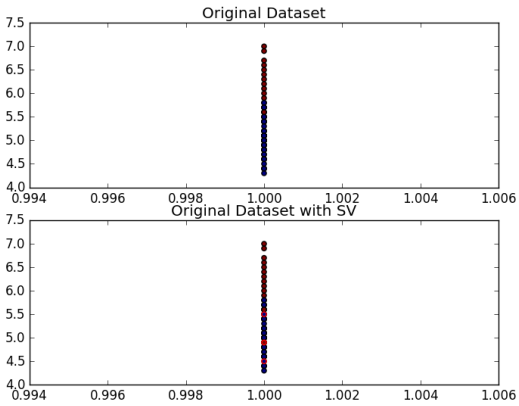
5. One class having less examples

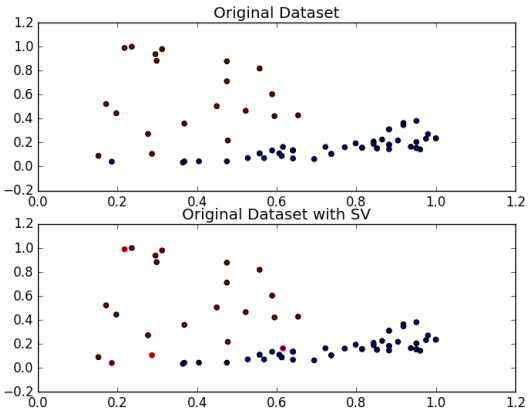
Iris (Separable Dataset)

In this experiment, I am using only 70 examples which contains only 20 examples of the other class. Below table shows the results for my and python's implementation which is same for all the 3 kernels. The reason behind is obvious since it's a separable dataset as we have seen above where there were more examples. The below graphs are for linear, polynomial degree 3 and radial kernels. Support Vectors can be identified in red colors.

Linear	*Python's Inbuilt Function*
Confusion Matrix: [[50 0] [0 20]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0)	Confusion Matrix: [[50 0] [0 20]] ('Accuracy: ', 1.0) ('Precision: ', 1.0) ('Recall: ', 1.0) ('F-Measure: ', 1.0)
Same for RBF and Polynomial Kernel	





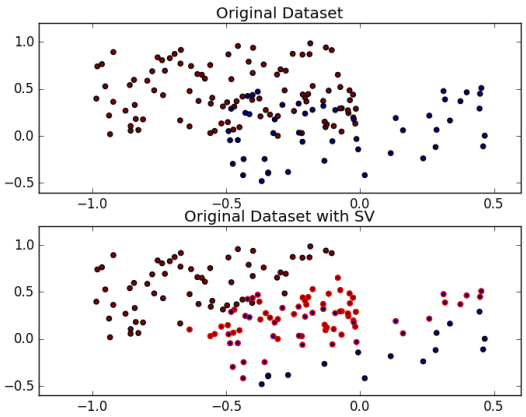


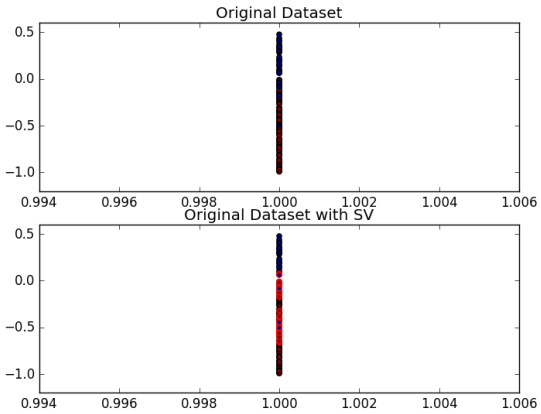
Generated Dataset (Non-Separable)

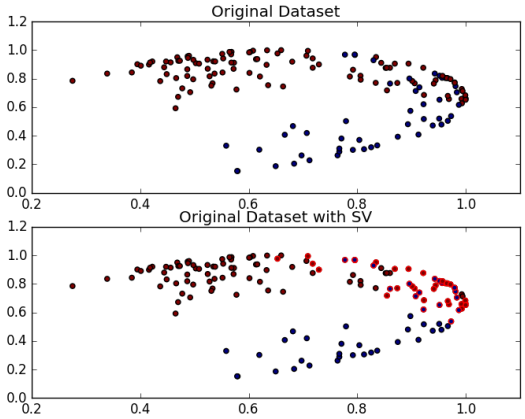
To compare the above results against non-separable datasets, I am using the one which I generated. From the below table we can see that in case of linear and radial the results are almost the same in both implementations, but in case of polynomial the python's inbuilt function wasn't able to classify one of the classes. The below graphs are for linear, polynomial degree 3 and radial kernels. Support Vectors can be identified in red colors.

Linear	*Polynomial*	*RBF*
Confusion Matrix:	Confusion Matrix:	Confusion Matrix:
[[43 7]	[[38 12]	[[36 14]
[5 95]]	[6 94]]	[1 99]]
('Accuracy: ', 0.92)	('Accuracy: ', 0.88)	('Accuracy: ', 0.90)
('Precision: ', 0.93)	('Precision: ', 0.89)	('Precision: ', 0.88)
('Recall: ', 0.94)	('Recall: ', 0.94)	('Recall: ', 0.99)
('F-Measure: ', 0.94)	('F-Measure: ', 0.91)	('F-Measure: ', 0.93)

Python's Inbuilt Function	***Python's Inbuilt Function***	***Python's Inbuilt Function***
Confusion Matrix: [[41 9] [3 97]] ('Accuracy: ', 0.92) ('Precision: ', 0.92) ('Recall: ', 0.97) ('F-Measure: ', 0.94)	Confusion Matrix: [[0 50] [0 100]] ('Accuracy: ', 0.67) ('Precision: ', 0.67) ('Recall: ', 1.0) ('F-Measure: ', 0.80)	Confusion Matrix: [[35 15] [0 100]] ('Accuracy: ', 0.90) ('Precision: ', 0.87) ('Recall: ', 1.0) ('F-Measure: ', 0.93)







References:

http://glowingpython.blogspot.com/2011/10/perceptron.html

http://scikit-learn.org/stable/auto_examples/ensemble/plot_voting_decision_regions.html

http://openclassroom.stanford.edu/MainFolder/DocumentPage.php?course=MachineLearning&doc=exercises/ex8/ex8.html

http://www.mblondel.org/journal/2010/09/19/support-vector-machines-in-python/

http://scikit-learn.org/stable/modules/svm.html

Note:

The derivation document is placed along with this report in the same folder.

All the codes have been executed and evaluated for the datasets mentioned above.

Kernel Implementation of SVM can be find in each of the codes mentioned as comments. For specific kernel, remove the comments and run the code.

To run the code successfully you need to change the file name and location in below codes, references are provided below.
Iris.py:13, Stanford.py:13