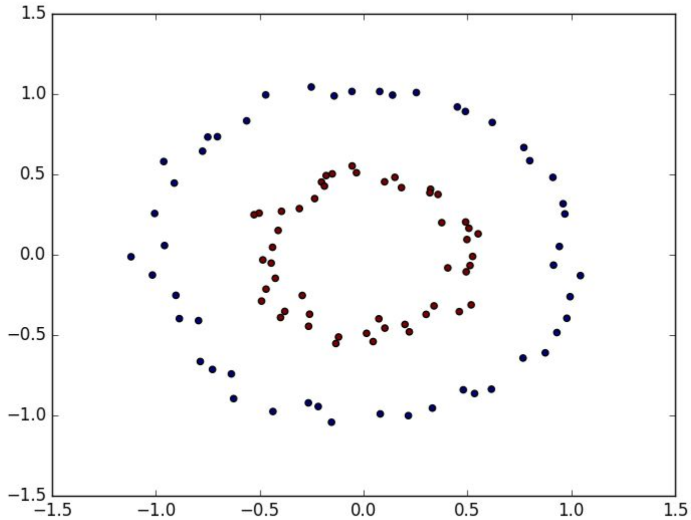
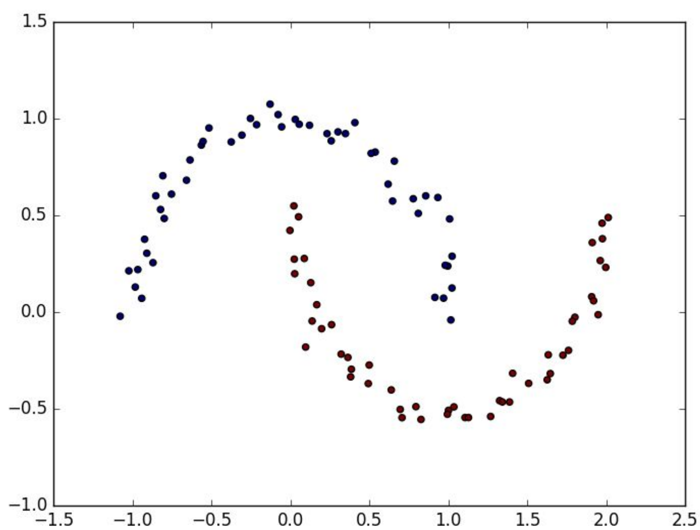
PLOTTING DATASETS

Plot for data_1:



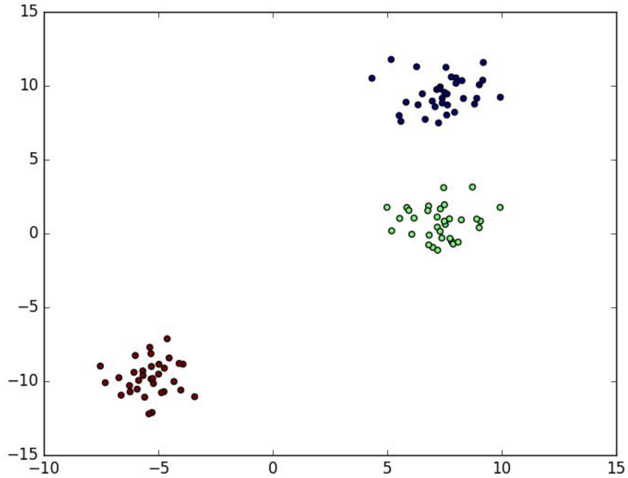
As we plot the datasets, they form 2 classes- as shown in red and blue. Clearly, it isn't possible to have a linear boundary.

Plot for data_2:



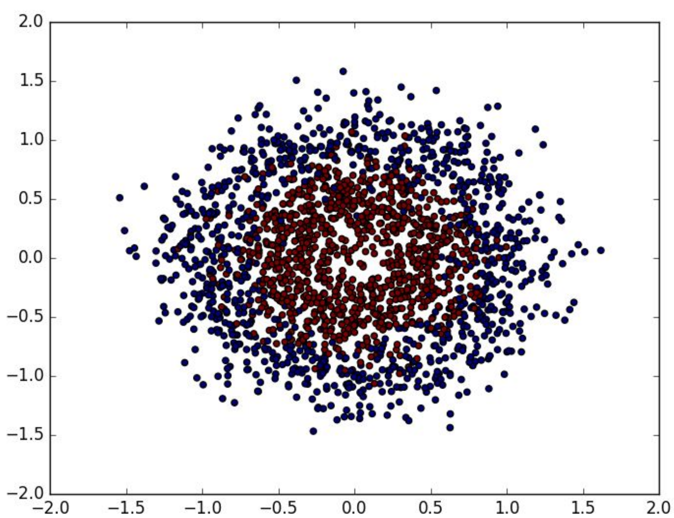
As we plot the datasets, they form 2 classes- as shown in red and blue. Clearly, it isn't possible to have a linear boundary.

Plot for data_3:



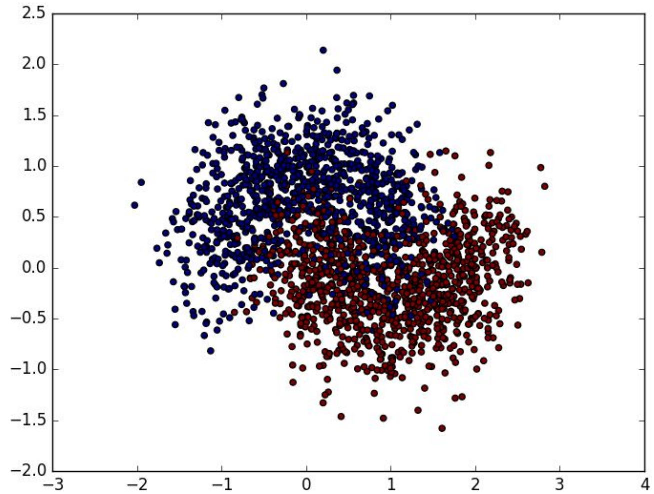
As we plot the datasets, they form 3 classes- as shown in red, green and blue. It is possible to have a linear boundary between the three classes.

Plot for data_4:



As we plot the datasets, they form 2 classes- as shown in red and blue. It is not possible to have a linear boundary between the two classes. Also there is a lot of noise.

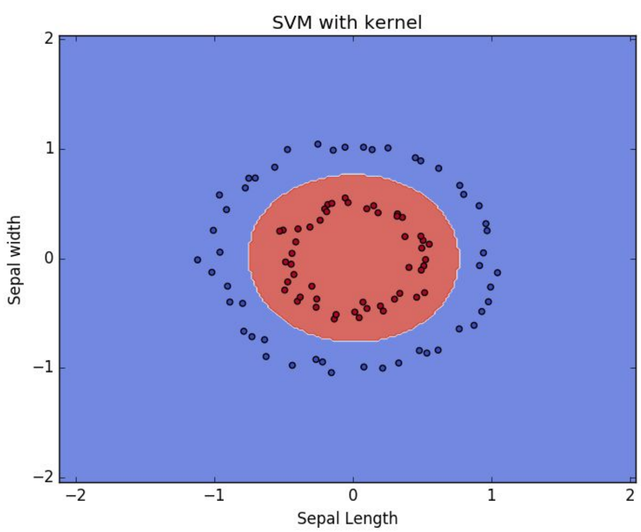
Plot for data_5:



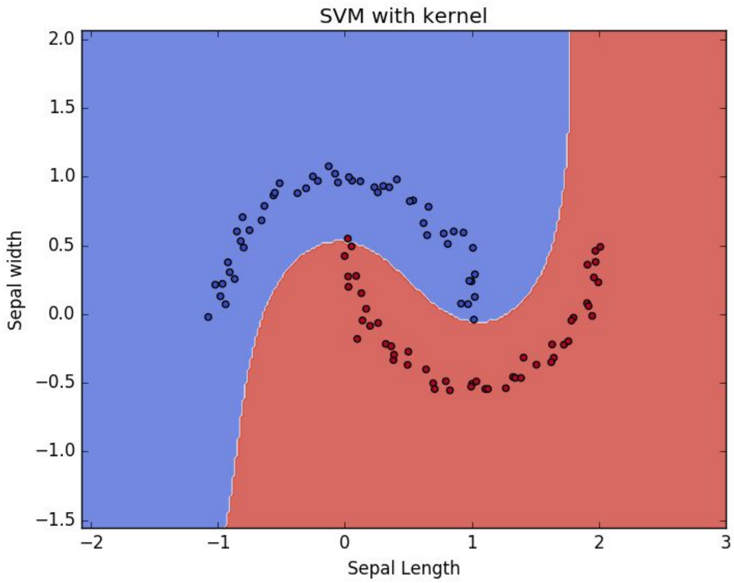
As we plot the datasets, they form 2 classes- as shown in red and blue. There is a lot of noise, not easy to tell about the boundary by just looking.

PLOTTING DATASETS WITH DECISION BOUNDARIES

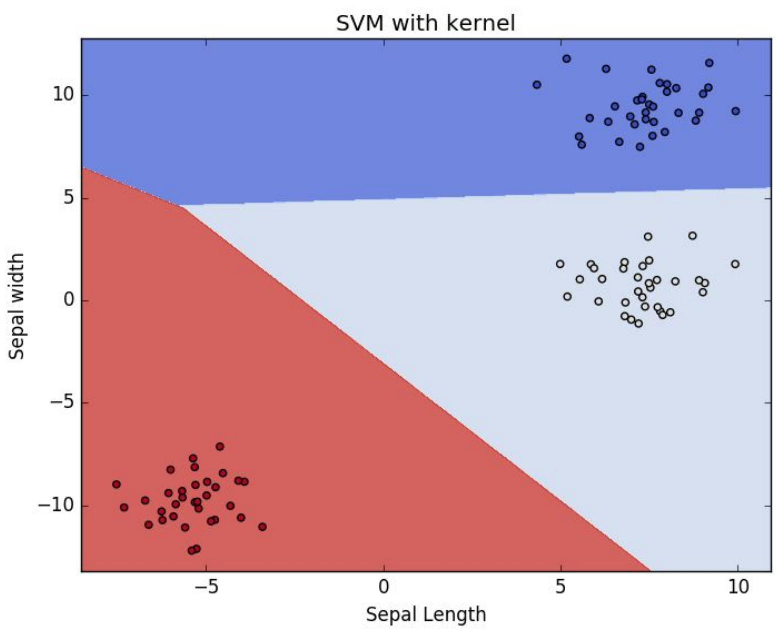
Plot for data_1:



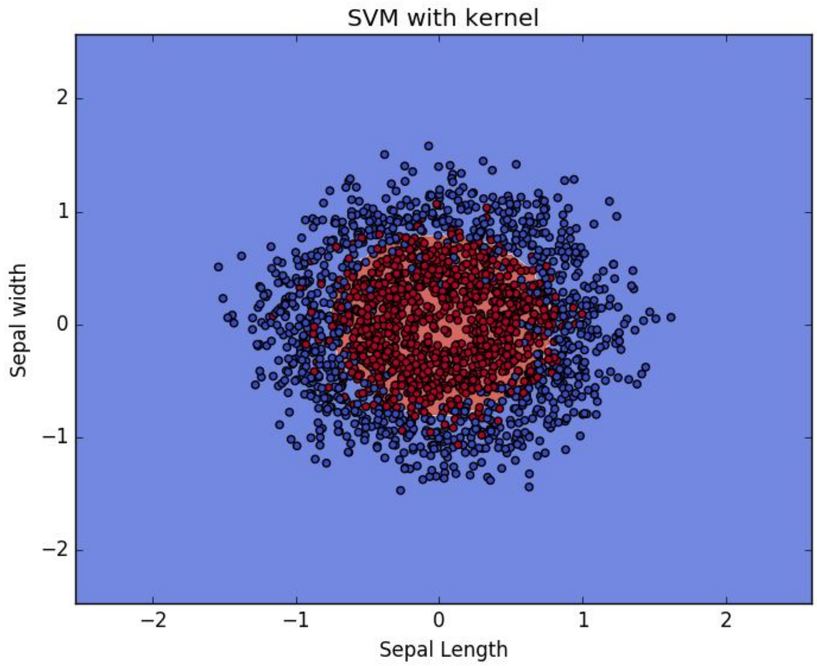
Plot for data_2:



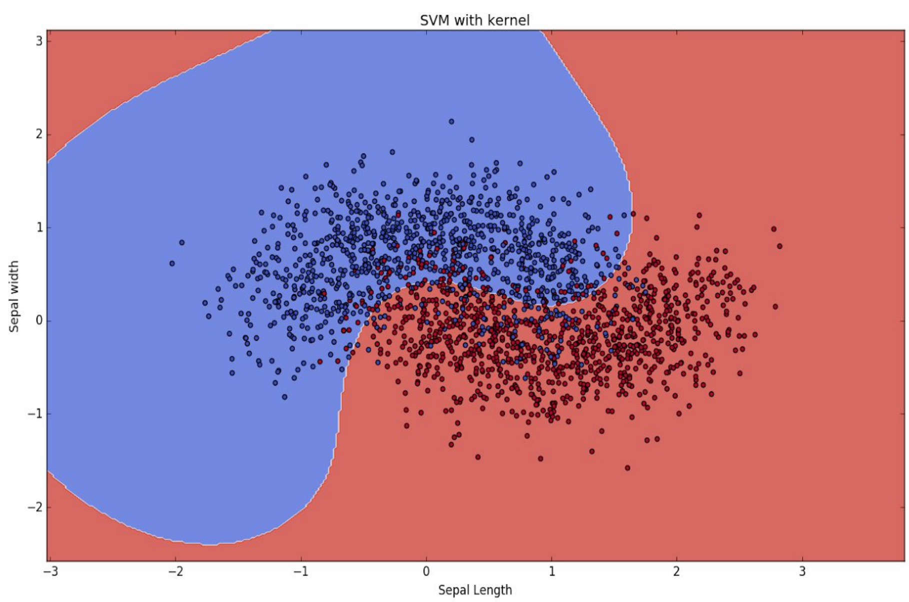
Plot for data_3:



Plot for data_4:

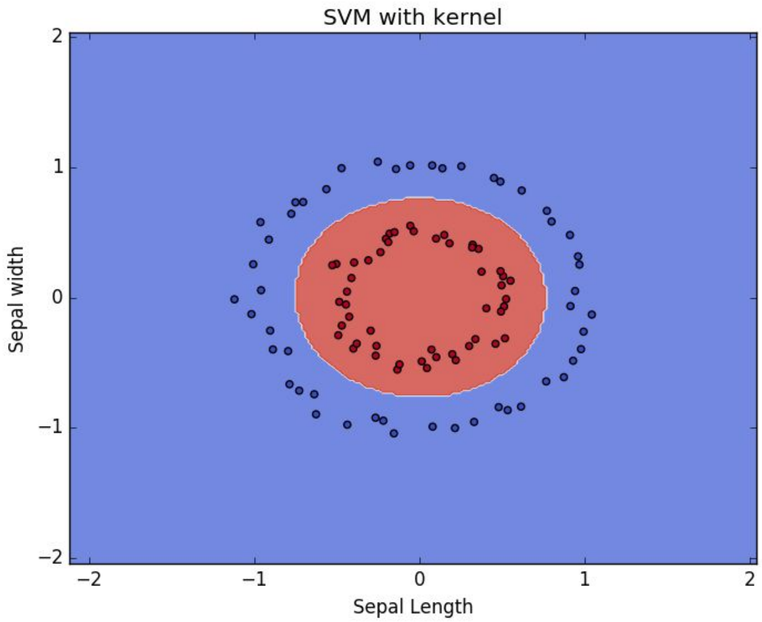


Plot for data_5:

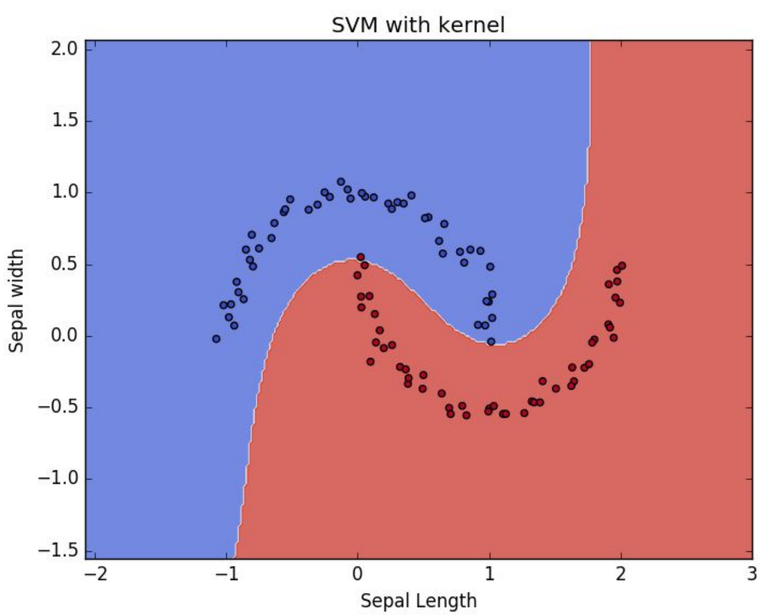


PLOTTING OUTLIER-REMOVED DATASETS

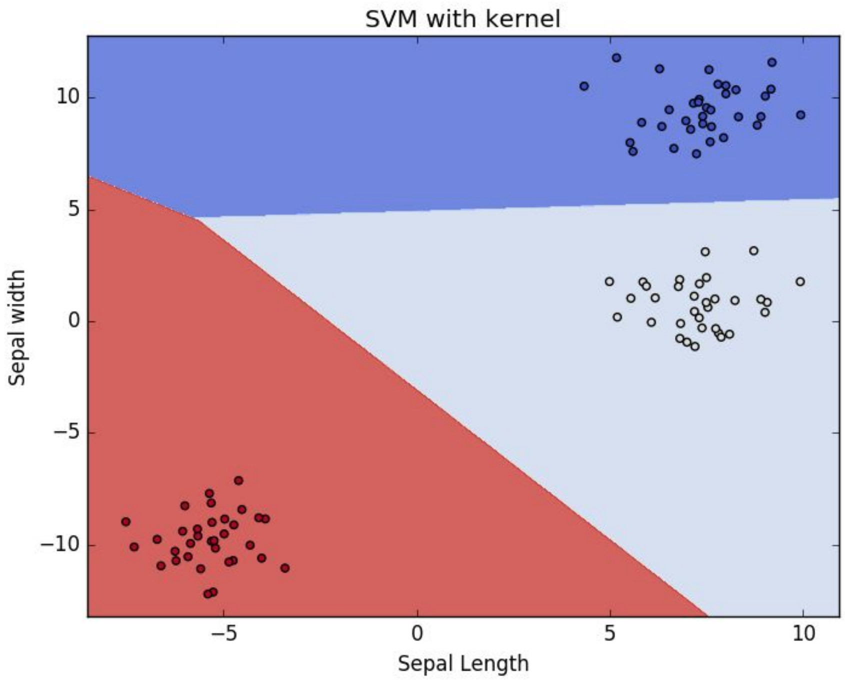
Plot for data_1: There were no outliers



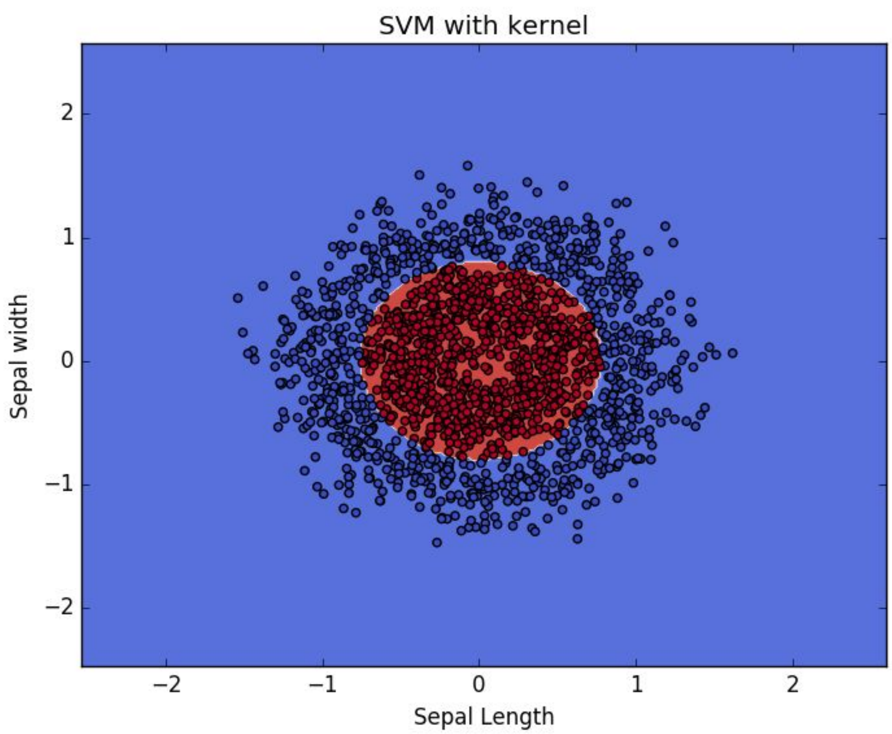
Plot for data_2: removed outliers.



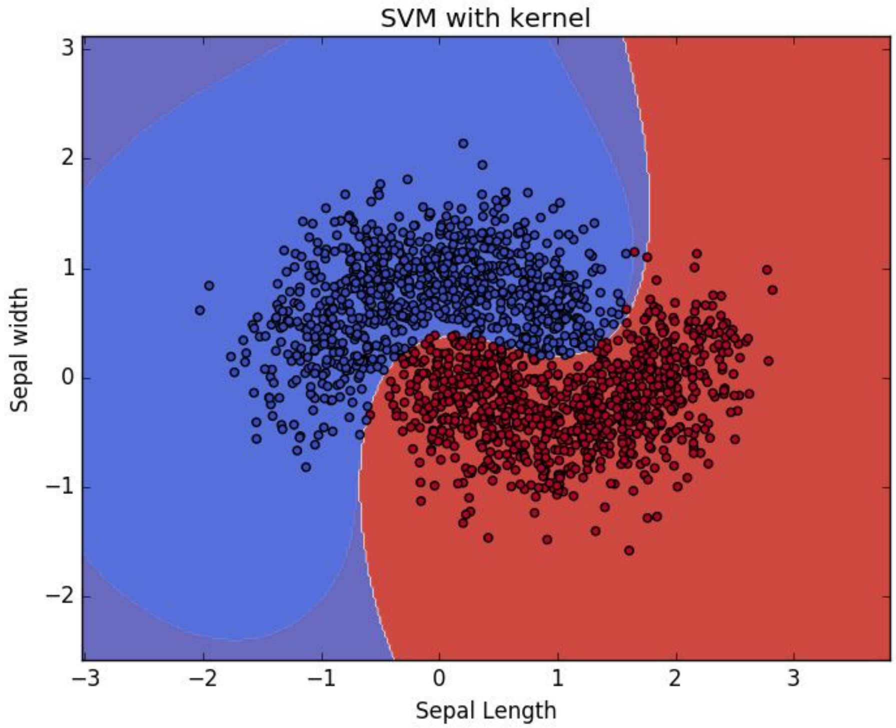
Plot for data_3: no outliers



Plot for data_4:



Plot for data_5:



PART-2 SVM

(Accuracy, value of C)

1. **Linear svm One vs One**:

data_1:

(0.43, 0.001)
(0.43, 0.1)
(0.39, 1.0)
(0.41, 10)
(0.41, 1000)

Data_2:
(0.58, 0.001)
(0.84, 0.1)
(0.85, 1.0)
(0.86, 10)
(0.86, 1000)

Data_3:
(1.0, 0.001)

(1.0, 0.1)
(1.0, 1.0)
(1.0, 10)
(1.0, 1000)

Data_4:
(0.486, 0.001)
(0.5175, 0.1)
(0.525, 1.0)
(0.5225, 10)
(0.522, 1000)

Data_5:
(0.78, 0.001)
(0.55, 0.1)
(0.51, 1.0)
(0.534, 10)
(0.58, 1000)

2. **Rbf svm One vs One**:
data_1:
(0.44, 0.001)
(0.59, 0.1)
(0.5, 1.0)
(0.5, 10)
(0.5, 1000)

Data2:
(0.55, 0.001)
(0.87, 0.1)
(0.45, 1.0)
(0.39, 10)
(0.36, 1000)

Data_3:
(0.33, 0.001)
(0.33, 0.1)
(0.33, 1.0)
(0.33, 10)
(0.33, 1000)

Data_4:
(0.486, 0.001)
(0.523, 0.1)
(0.5285, 1.0)
(0.5285, 10)
(0.56,1000)

Data_5:
(0.5, 0.001)
(0.51, 0.1)
(0.525, 1.0)
(0.52, 10)
(0.52, 1000)

3. **Linear svm one vs rest:**
data_1:
(0.43, 0.001)
(0.43, 0.1)
(0.39, 1.0)
(0.41, 10)
(0.41, 1000)

Data_2:
(0.58, 0.001)
(0.84, 0.1)
(0.85, 1.0)
(0.86, 10)
(0.86, 1000)

Data_3:
(0.92, 0.001)
(1.0, 0.1)
(1.0, 1.0)
(0.99, 10)
(0.98, 1000)

Data_4:
(0.486, 0.001)
(0.5175, 0.1)
(0.525, 1.0)

(0.522, 10)
(0.522, 1000)

Data_5:
(0.52, 0.001)
(0.5532, 0.1)
(0.57, 1.0)
(0.578, 10)
(0.57, 1000)

4. **Rbf svm one vs rest:**
data_1:
(0.44, 0.001)
(0.6, 0.1)
(0.5, 1.0)
(0.5, 10)
(0.5, 1000)

Data_2:
(0.55, 0.001)
(0.87, 0.1)
(0.45, 1.0)
(0.39, 10)
(0.36, 1000)

Data_3:
(0.33, 0.001)
(0.33, 0.1)
(0.33, 1.0)
(0.33, 10)
(0.4, 1000)

Data_4:
(0.23, 0.001)
(0.59, 0.1)
(0.66, 1.0)
(0.7, 10)
(0.7,1000)

Data_5:
(0.412, 0.001)

(0.52, 0.1)
(0.509, 1.0)
(0.52, 10)
(0.513, 1000)

Observations:
What choice of hyperparameters is useful for which dataset and why? How did you choose the hyperparameters?
The hyperparameters were chosen using grid search.
Dataset 1 and 2 have minimum or no noise, hence the optimal value for C for them is 0.1.
Dataset 3 is already linearly separable, hence linear kernel worked better for it! C again, 0.1 was good and higher values also gave good results because the data was not at all noisy, so harder the margin, better is the boundary.
Dataset 4 and 5 were very noisy. As we got to higher values of c, we achieved better accuracy, because then the boundary got more strict.
The previous datasets have a linear boundary possible except the B part. Hence, for A and C we prefer linear and for B part we prefer rbf. We should keep value of C = 0.1 as these are multidimesnional data and we want to avoid having a large value of C as it will cause overfitting.
The confusion matrices produced are of this sort:

```
[[ 1.  0.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
```

```
[ 0.  1.]
[ 0.  1.]
[ 1.  0.]
[ 1.  0.]
[ 0.  1.]
[ 1.  0.]
[ 0.  1.]
[ 0.  1.]
[ 0.  1.]
[ 0.  1.]
[ 1.  0.]
[ 1.  0.]
[ 0.  1.]
[ 0.  1.]
[ 1.  0.]
[ 0.  1.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 1.  0.]
[ 0.  1.]
[ 0.  1.]
[ 0.  1.]
[ 1.  0.]
[ 0.  1.]
```

```
 [ 1.  0.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 1.  0.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]
 [ 1.  0.]
 [ 0.  1.]
 [ 0.  1.]]
```

And another example: (depending on the classes and sample size)

```
[[ 0.  1.  2.]
 [ 0.  1.  2.]
 [ 0.  1.  2.]
 [ 2.  1.  0.]
 [ 0.  1.  2.]
 [ 2.  1.  0.]
 [ 0.  1.  2.]
 [ 0.  1.  2.]
```

```
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 2.  1.  0.]
```

```
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 2.  1.  0.]
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 1.  2.  0.]
```

```
[ 0.  1.  2.]
[ 1.  2.  0.]
[ 1.  2.  0.]
[ 0.  1.  2.]
[ 2.  1.  0.]
[ 1.  2.  0.]]
```

3. Kaggle question

Dataset:
Train.json: It had 472426 entries. All with outliers and noise included.
Test.json: It had 472426 entries. We had to predict y.

Preprocessing and techniques used: *tfidf vectorisation, outlier removal(as used in part 1 of the assignment), linear svc to fit the data*.