# ELL409 Assignment 2 Report

Kshitij Alwadhi (2019EE10577)

22nd October 2021

## 1. Introduction

In this assignment, we experiment with the use of SVMs for both binary and multiclass classification problems, and understand the effects of varying various hyperparameters therein.

## 2. Part 1A

### 2.1 Formulating the problem for CVXOPT.

We have the following optimization problem in the case of an SVM with L1 regularization:

$$\min_{\gamma, w, b} \quad \frac{1}{2}\|w\|^2 + C \cdot \sum_{i=1}^{m} \xi_i$$

$$\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i \quad , \; i = 1 \ldots m$$

$$\xi_i \geq 0$$

To solve this problem, we take the help of Lagrangian Multipliers and proceed using KKT conditions. (The following results are taken from the **CS229 SVM notes**)

The Lagrangian for the optimization problem is:

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2} w^T w + C \cdot \sum_{i=1}^{m} \xi_i - \sum_{i=1}^{m} \alpha_i \left[ y^{(i)}(x^T w + b) - 1 + \xi_i \right] - \sum_{i=1}^{m} r_i \xi_i$$

After applying the KKT conditions (taking the respective partial derivatives), we get the following optimization problem:

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} \cdot y^{(j)} \cdot \alpha_i \cdot \alpha_j \cdot \langle x^{(i)}, x^{(j)} \rangle$$

$$\langle x^{(i)}, x^{(j)} \rangle \rightarrow K(x^{(i)}, x^{(j)})$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C \quad , \quad i = 1 \ldots m$$

$$\sum_{i=1}^{m} \alpha_i \cdot y^{(i)} = 0$$

This is equivalent to the following minimization problem:

$$\min_{\alpha} \quad \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} \cdot y^{(j)} \cdot \alpha_i \cdot \alpha_j \cdot K(x^{(i)}, x^{(j)}) - \sum_{i=1}^{m} \alpha_i$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C \quad , \quad i = 1 \ldots m$$

$$\sum_{i=1}^{m} \alpha_i \cdot y^{(i)} = 0$$

To simplify things, we define the following:

We define a matrix $H$ such that

$$H_{ij} = y^{(i)} \cdot y^{(j)} \cdot K(x^{(i)}, x^{(j)})$$

Now, our optimization problem looks like:

$$\min_{\alpha} \quad \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \cdot \alpha_j \cdot H_{ij} - \sum_{i=1}^{m} \alpha_i$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C \quad , \quad i = 1 \ldots m$$

$$\sum_{i=1}^{m} \alpha_i \cdot y^{(i)} = 0$$

Now this is a quadratic optimization problem but we need to convert it into a form which we can feed into CVXOPT.

CVX opt general form for QP:

$$\min_{x} \quad \frac{1}{2} x^T P x + q^T x$$

$$\text{s.t.} \quad Gx \preceq h$$

$$Ax = b$$

$$x = \text{cvxopt.solvers.qp}(P, q, G, h, A, b)$$

We can convert our optimization problem to the following form so that it resembles the form of QP in CVXOPT:

$$\min_{\alpha} \quad \frac{1}{2} \alpha^T H \alpha + [-1 \; -1 \; \cdots \; -1] \alpha$$

$$\text{s.t.} \quad \begin{bmatrix} -I \\ I \end{bmatrix}_{n \times n} \alpha \preceq \begin{bmatrix} 0 \\ C \end{bmatrix}_{n \times 1}$$

$$y^T \alpha = 0$$

On comparing the terms, we have the following:

$$P = H \qquad q = \begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$$

$$G = \begin{bmatrix} -I \\ I \end{bmatrix} \quad \begin{matrix} \rightarrow n \times n \\ \rightarrow n \times n \end{matrix} \qquad h = \begin{bmatrix} 0 \\ C \end{bmatrix} \quad \begin{matrix} \rightarrow n \times 1 \\ \rightarrow n \times 1 \end{matrix}$$

$$A = y^T \qquad b = 0$$

This can be solved using CVXOPT by the following:

```
α = cvxopt.solvers.qp(P, q, G, h, A, b)
```

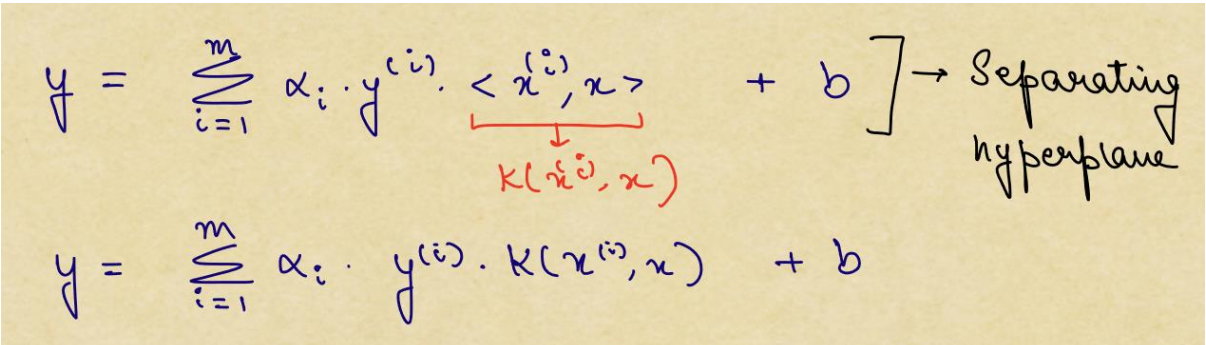
The following is the code which solves for α using CVXOPT:

```
        X = np.array(X)
        y = np.array(y)
        num_samples,num_features = X.shape
        K = np.zeros((num_samples,num_samples))

        for i in (range(num_samples)):
            for j in range(num_samples):
                if self.kernel == 'LINEAR':
                    K[i][j] = np.dot(X[i],np.transpose(X[j]))
                elif self.kernel == 'POLY':
                    K[i][j] = (np.dot(X[i],np.transpose(X[j])) + self.coeff) **␣
↪self.power
                elif self.kernel == 'RBF':
                    K[i][j] = np.exp(-1 * self.gamma*np.sum(np.
↪square(X[i]-X[j])))

        self.K = K
```

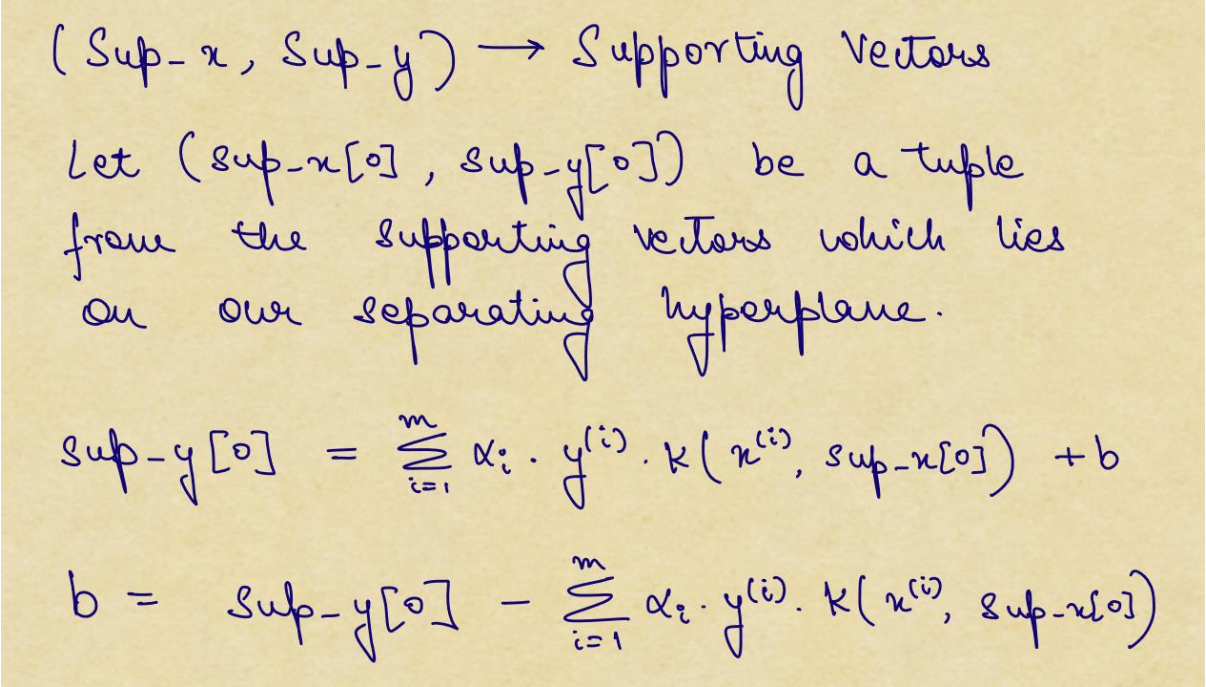
```
            H = np.zeros((num_samples,num_samples))
            for i in (range(num_samples)):
                for j in range(num_samples):
                    H[i][j] = y[i]*y[j] * K[i][j]
            P = matrix(H)
            q = matrix(np.ones(num_samples) * -1)
            G = matrix(np.vstack(((np.identity(num_samples) * -1),np.
↪identity(num_samples))))
            h = matrix(np.hstack((np.zeros(num_samples),np.
↪ones(num_samples)*self.C)))
            A = matrix(y,(1,num_samples))
            b = matrix(0.0)

            solvers.options['show_progress'] = False
            soln = solvers.qp(P,q,G,h,A,b)
            alpha = np.array(soln['x'])
```

Now that we have solved for the value of α, we need to determine our supporting vectors, store the non-zero α values and also calculate the value of b so that we can make predictions.

The equation of separating hyperplane is given by:



We know that the supporting vectors lie on the separating hyperplanes, so we exploit that and write the following:




We find the value of b using the above formula. Also, we can directly find the indices of the supporting vectors by checking where the indices of α are non-zero. These steps are performed using the following code:

```
            self.sup_idx = np.where(alpha>1e-5)[0]
            self.ind =  np.arange(len(alpha))[self.sup_idx]

            self.sup_x = X[self.sup_idx,:]
            self.sup_y = y[self.sup_idx]
            self.alpha = alpha[self.sup_idx]
            self.b = self.sup_y[0]

            for i in range(len(self.alpha)):
                if self.kernel == 'LINEAR':
                    temp = np.dot(self.sup_x[i],np.transpose(self.sup_x[0]))
                elif self.kernel == 'POLY':
                    temp = (np.dot(self.sup_x[i],np.transpose(self.
↪sup_x[0]))+self.coeff)**self.power
                elif self.kernel == 'RBF':
                    temp = np.exp(-1 * self.gamma*np.sum(np.square(self.
↪sup_x[i]-self.sup_x[0])))
                self.b -= self.alpha[i] * self.sup_y[i] * temp
```

At a later stage, I made a change to the value of b since I was getting slightly different results in some cases (reference: stack exchange link in reference):

```
            self.b = 0
            for i in range(len(self.alpha)):
                self.b += self.sup_y[i]
                self.b -= np.sum(self.alpha * self.sup_y * self.K[self.
↪ind[i]][self.sup_idx])
            self.b /= len(self.alpha)
            self.sup_idx = np.where(self.sup_idx == True)[0]
```

For deciding the class of the input X, we see which side of the separating hyperplane our data point is and give it a +1 class if it's above the hyperplane and -1 if it's below the hyperplane. This is performed by the following code snippet:

```
  def predict(self,X):
      preds = []
      for x in X:
          pred = 0
          for i in range(len(self.alpha)):
              if self.kernel == 'LINEAR':
                  temp = np.dot(self.sup_x[i],np.transpose(x))
              elif self.kernel == 'POLY':
                  temp = (np.dot(self.sup_x[i],np.transpose(x)) + self.coeff)␣
↪** self.power
              elif self.kernel == 'RBF':
                  temp = np.exp(-1 * self.gamma *np.sum(np.square(self.
↪sup_x[i]-x)))
              pred += self.alpha[i] * self.sup_y[i] * temp
          pred += self.b
          if pred>=0:
              preds.append(1.0)
          else:
              preds.append(-1.0)
      return np.array(preds)
```

We also compare the results we get using the optimization library with the standard SVM library (LIBSVM). Our setup will be the parameters obtained in Binary Classification section of the assignment as follows **(also, we use 5-fold cross validation for our experiments)**:

| | Class C1,C2 | | |
|---|---|---|---|
| | (1,8) | (3,7) | (4,9) |
| Linear (C) | 0.1 | 0.01 | 0.1 |
| RBF (C) | 1 | 0.1 | 1 |
| RBF (γ) | 0.01 | 0.1 | 0.1 |

I. C1 = 1, C2 = 8 (Linear Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 98.0241935483871 | 97.25806451612902 | 00.160021 |
| CVXOPT | 97.94354838709677 | 97.09677419354838 | 06.822120 |

II. C1 = 1, C2 = 8 (RBF Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 98.30645161290322 | 98.06451612903224 | 00.171265 |
| CVXOPT | 98.46774193548387 | 98.06451612903224 | 14.654499 |

III. C1 = 4, C2 = 9 (Linear Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 97.0796460176991 | 96.10619469026547 | 00.150144 |
| CVXOPT | 96.99115044247787 | 95.75221238938052 | 06.162015 |

IV. C1 = 4, C2 = 9 (RBF Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 99.64601769911505 | 97.34513274336283 | 00.219928 |
| CVXOPT | 99.69026548672568 | 97.16814159292035 | 19.763364 |

We also compare the support vectors obtained using the two approaches:

## I. C1 = 1, C2 = 8 (Linear Kernel)

```
Mode: LIBSVM
Training Accuracy: 97.63948497854076
Testing Accuracy: 97.43589743589743
63 number of support vectors found
The indices of the support vectors are: [5, 11, 13, 15, 22, 31, 36, 40, 44, 46, 56, 60, 63, 76, 78, 80, 86, 91, 111, 115, 118, 1
38, 139, 143, 148, 151, 153, 161, 168, 172, 182, 210, 221, 236, 242, 245, 246, 259, 281, 293, 300, 305, 307, 310, 317, 322, 339,
342, 343, 345, 351, 356, 359, 360, 367, 372, 394, 395, 398, 399, 401, 417, 462]

Mode: CVXOPT
Training Accuracy: 97.63948497854076
Testing Accuracy: 97.43589743589743
63 number of support vectors found
The indices of the support vectors are: [5, 11, 13, 15, 22, 31, 36, 40, 44, 46, 56, 60, 63, 76, 78, 80, 86, 91, 111, 115, 118, 1
38, 139, 143, 148, 151, 153, 161, 168, 172, 182, 210, 221, 236, 242, 245, 246, 259, 281, 293, 300, 305, 307, 310, 317, 322, 339,
342, 343, 345, 351, 356, 359, 360, 367, 372, 394, 395, 398, 399, 401, 417, 462]
```

## II. C1 = 1, C2 = 8 (RBF Kernel)

```
Mode: LIBSVM
Training Accuracy: 98.49785407725322
Testing Accuracy: 96.15384615384616
38 number of support vectors found
The indices of the support vectors are: [5, 13, 17, 21, 31, 36, 46, 56, 60, 78, 80, 86, 91, 111, 115, 118, 119, 143, 148, 161, 1
68, 182, 221, 245, 259, 281, 317, 339, 342, 343, 345, 351, 352, 359, 372, 395, 399, 401]

Mode: CVXOPT
Training Accuracy: 98.49785407725322
Testing Accuracy: 96.15384615384616
38 number of support vectors found
The indices of the support vectors are: [5, 13, 17, 21, 31, 36, 46, 56, 60, 78, 80, 86, 91, 111, 115, 118, 119, 143, 148, 161, 1
68, 182, 221, 245, 259, 281, 317, 339, 342, 343, 345, 351, 352, 359, 372, 395, 399, 401]
```

## III. C1 = 4, C2 = 9 (Linear Kernel)

```
Mode: LIBSVM
Training Accuracy: 96.93396226415094
Testing Accuracy: 96.47887323943662
89 number of support vectors found
The indices of the support vectors are: [2, 3, 5, 8, 12, 13, 14, 15, 16, 18, 21, 24, 26, 27, 30, 33, 36, 50, 56, 58, 64, 73, 74,
80, 81, 83, 93, 103, 104, 110, 123, 125, 136, 137, 144, 148, 157, 166, 167, 168, 170, 173, 174, 175, 177, 178, 179, 188, 196, 19
7, 205, 214, 219, 232, 233, 242, 244, 246, 255, 257, 261, 262, 266, 272, 275, 289, 293, 295, 306, 307, 318, 334, 336, 341, 344,
349, 358, 360, 364, 370, 382, 387, 389, 390, 394, 396, 406, 409, 410]

Mode: CVXOPT
Training Accuracy: 97.16981132075472
Testing Accuracy: 97.1830985915493
89 number of support vectors found
The indices of the support vectors are: [2, 3, 5, 8, 12, 13, 14, 15, 16, 18, 21, 24, 26, 27, 30, 33, 36, 50, 56, 58, 64, 73, 74,
80, 81, 83, 93, 103, 104, 110, 123, 125, 136, 137, 144, 148, 157, 166, 167, 168, 170, 173, 174, 175, 177, 178, 179, 188, 196, 19
7, 205, 214, 219, 232, 233, 242, 244, 246, 255, 257, 261, 262, 266, 272, 275, 289, 293, 295, 306, 307, 318, 334, 336, 341, 344,
349, 358, 360, 364, 370, 382, 387, 389, 390, 394, 396, 406, 409, 410]
```

## IV. C1 = 4, C2 = 9 (RBF Kernel)

```
Mode: LIBSVM
Training Accuracy: 99.76415094339622
Testing Accuracy: 97.1830985915493
319 number of support vectors found
The indices of the support vectors are: [0, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 35, 36, 38, 39, 40, 42, 43, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65,
66, 67, 69, 70, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 98, 100, 101, 103, 104,
105, 107, 109, 110, 112, 114, 115, 116, 117, 119, 120, 121, 123, 124, 125, 126, 127, 129, 130, 131, 133, 134, 136, 138, 141, 14
3, 144, 145, 147, 148, 149, 151, 152, 153, 154, 155, 156, 157, 159, 161, 163, 164, 165, 166, 167, 168, 170, 172, 173, 174, 175,
177, 178, 179, 180, 182, 183, 185, 186, 187, 188, 189, 193, 194, 195, 196, 197, 198, 201, 202, 205, 206, 208, 210, 211, 212, 21
3, 214, 215, 217, 219, 220, 221, 223, 225, 226, 227, 228, 230, 232, 233, 235, 236, 237, 238, 240, 242, 244, 246, 251, 252, 253,
255, 256, 257, 258, 259, 261, 262, 263, 266, 267, 268, 270, 272, 273, 274, 275, 276, 278, 281, 282, 283, 284, 287, 289, 290, 29
2, 293, 294, 295, 297, 298, 299, 300, 302, 304, 305, 306, 307, 310, 311, 312, 313, 315, 318, 319, 320, 321, 322, 323, 326, 327,
328, 329, 330, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 35
5, 356, 358, 359, 360, 361, 362, 363, 364, 365, 367, 368, 369, 370, 372, 374, 375, 376, 377, 378, 379, 381, 382, 384, 387, 388,
389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 402, 403, 405, 406, 407, 408, 409, 410, 412, 413, 414, 416, 420, 42
1, 422, 423]

Mode: CVXOPT
Training Accuracy: 99.76415094339622
Testing Accuracy: 97.1830985915493
330 number of support vectors found
The indices of the support vectors are: [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 35, 36, 38, 39, 40, 42, 43, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64,
65, 66, 67, 69, 70, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 98, 100, 101, 103, 1
04, 105, 107, 109, 110, 112, 114, 115, 116, 117, 119, 120, 121, 123, 124, 125, 126, 127, 129, 130, 131, 133, 134, 136, 137, 138,
139, 141, 143, 144, 145, 147, 148, 149, 151, 152, 153, 154, 155, 156, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 170, 17
2, 173, 174, 175, 177, 178, 179, 180, 182, 183, 185, 186, 187, 188, 189, 193, 194, 195, 196, 197, 198, 201, 202, 205, 206, 208,
210, 211, 212, 213, 214, 215, 217, 219, 220, 221, 223, 225, 226, 227, 228, 230, 232, 233, 235, 236, 237, 238, 240, 242, 244, 24
6, 251, 252, 253, 255, 256, 257, 258, 259, 261, 262, 263, 266, 267, 268, 269, 270, 272, 273, 274, 275, 276, 278, 279, 281, 282,
283, 284, 287, 289, 290, 292, 293, 294, 295, 297, 298, 299, 300, 302, 304, 305, 306, 307, 310, 311, 312, 313, 315, 318, 319, 32
0, 321, 322, 323, 326, 327, 328, 329, 330, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349,
350, 351, 352, 353, 354, 355, 356, 358, 359, 360, 361, 362, 363, 364, 365, 367, 368, 369, 370, 371, 372, 374, 375, 376, 377, 37
8, 379, 380, 381, 382, 384, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 402, 403, 405, 406, 407,
408, 409, 410, 412, 413, 414, 416, 417, 419, 420, 421, 422, 423]
```

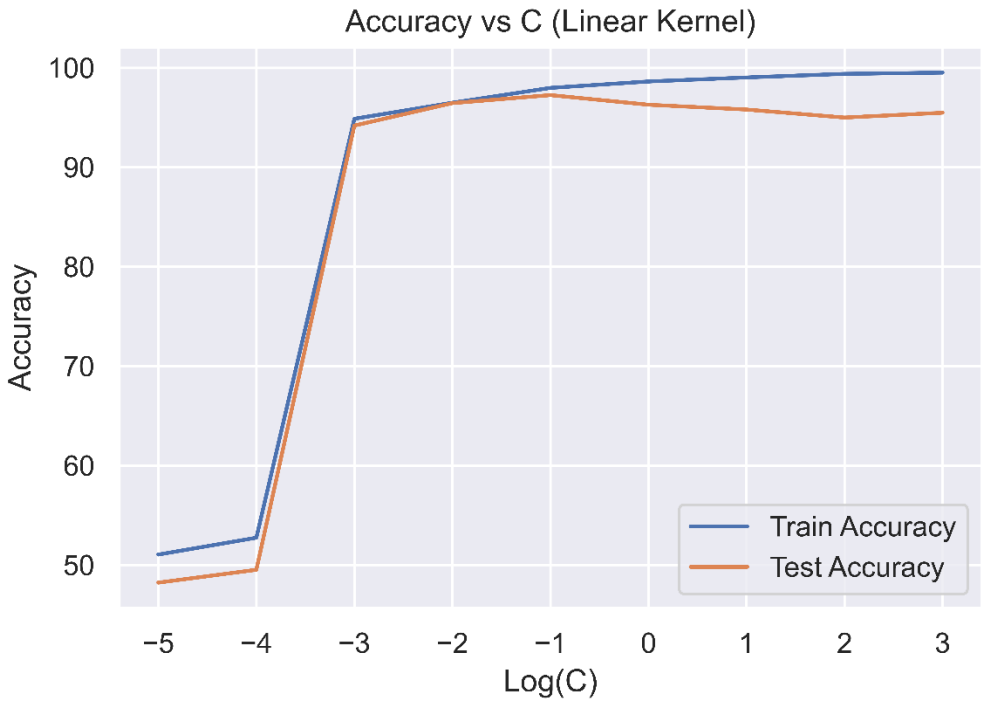
We observe that, in all the cases, the performance of the CVXOPT approach is similar to the LIBSVM approach and also, the support vectors are also same (or very similar) to those obtained using SVC from sklearn library (a wrapper for LIBSVM).

## 2.2 Binary Classification

In all of the analysis, I have used 5-fold Cross Validation and then averaged out the train and test accuracy for comparing the results of the various hyperparameter settings.

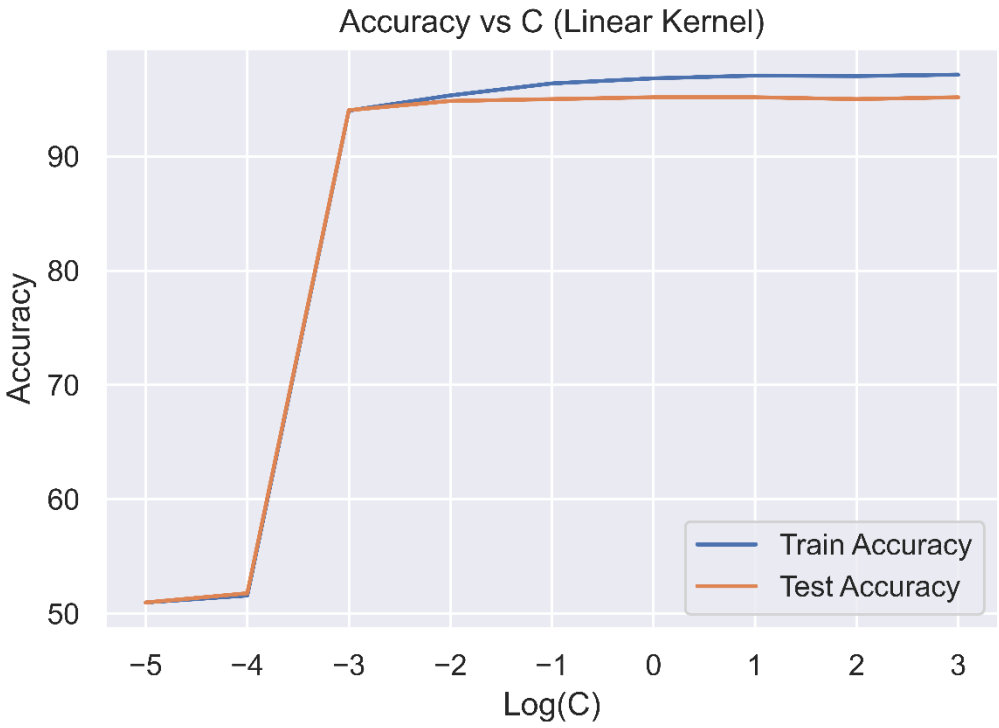
### 2.2.1 First we do the analysis for: **C1 = 1, C2 = 8**

Using **Linear Kernel** we get the following results:




(Accuracy vs C (Linear Kernel) for 25 features)

From the above plot, we can see that when the value of C is very low (<0.001), the model is underfitting the data but as the value of C increases, the model starts to fit the data (good fit) and we observe a peak Test accuracy at C = 0.1. Beyond this (C>0.1), we start to see that training accuracy creeps up to almost 100% and the test accuracy starts falling, hence indicating overfitting.



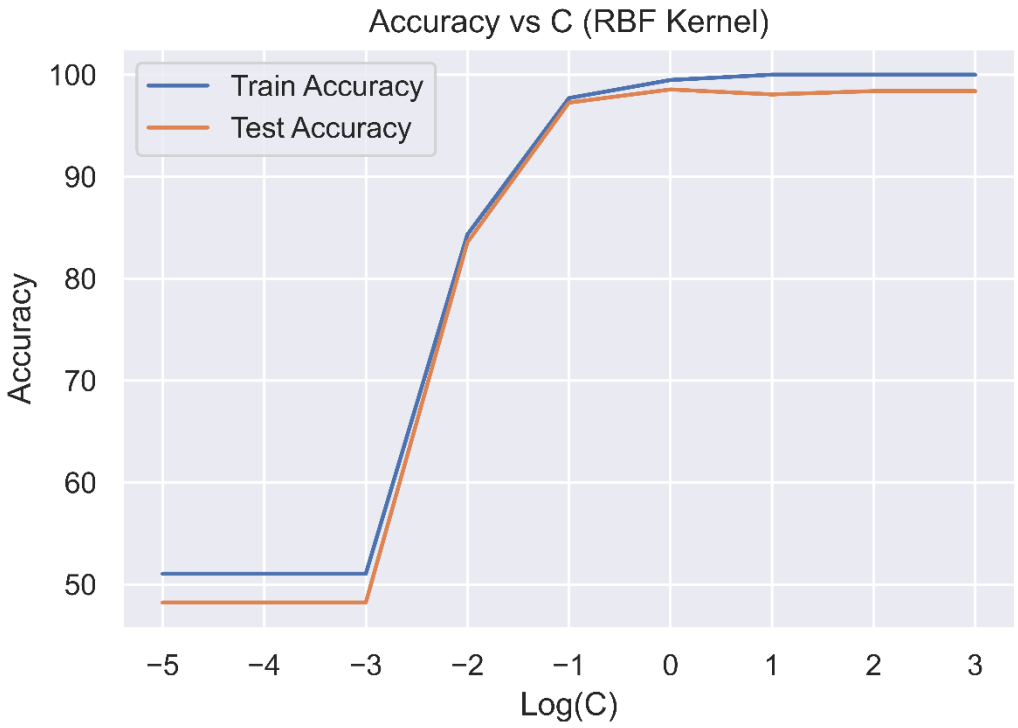

(Accuracy vs C (Linear Kernel) for 10 features)

Now, if we use only 10 features, we get a similar story, for $C < 0.001$, the model is underfitting the data and we get peak test accuracy at $C = 0.01$ beyond which the model starts overfitting the data.

There are two differences we observe here:

1) The peak test accuracy that we obtain when we use all 25 features is greater than when we use only 10 features.
2) The model is able to get a 100% train accuracy in the case of 25 features when C is significantly larger, however, this is not achievable in the case of 10 feature dataset.

Using **RBF kernel** we get the following results:

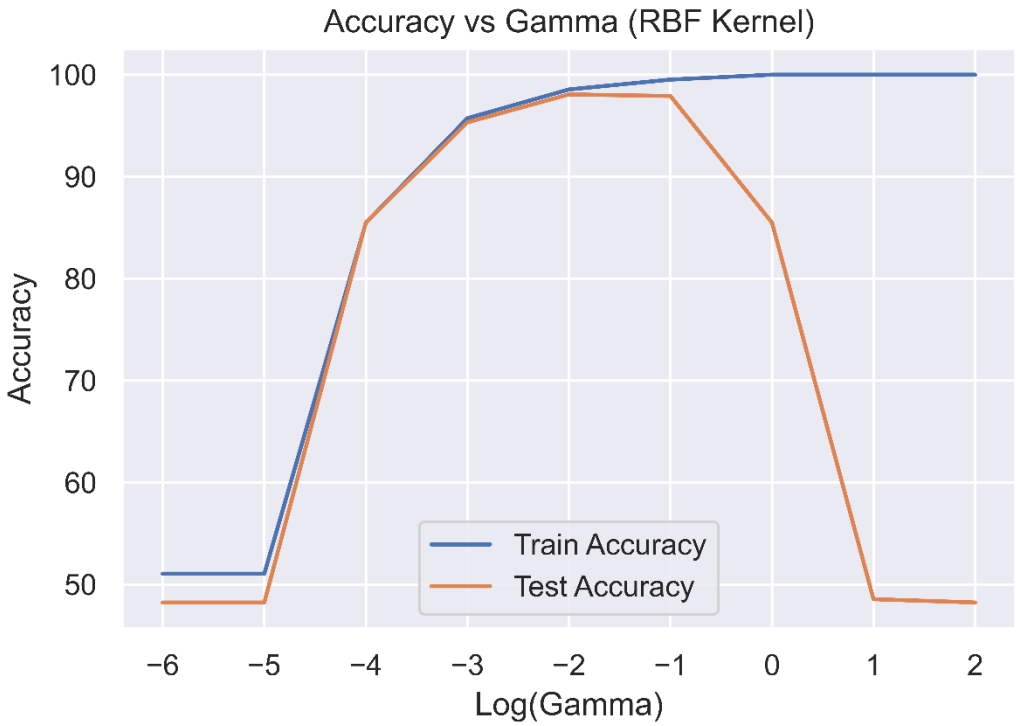



(Accuracy vs C (RBF Kernel) for 25 features)

In the case of RBF kernel, we have two parameters to play around with. For finding the appropriate value of C, we perform a logarithmic sweep while keeping the value of γ to be $\frac{1}{\#\ of\ features}$ as usually done by standard SVM libraries.

We observe the following:

| | |
|---|---|
| C < 1 | Underfitting |
| C = 1 | Best fit |
| C > 1 | Overfitting |

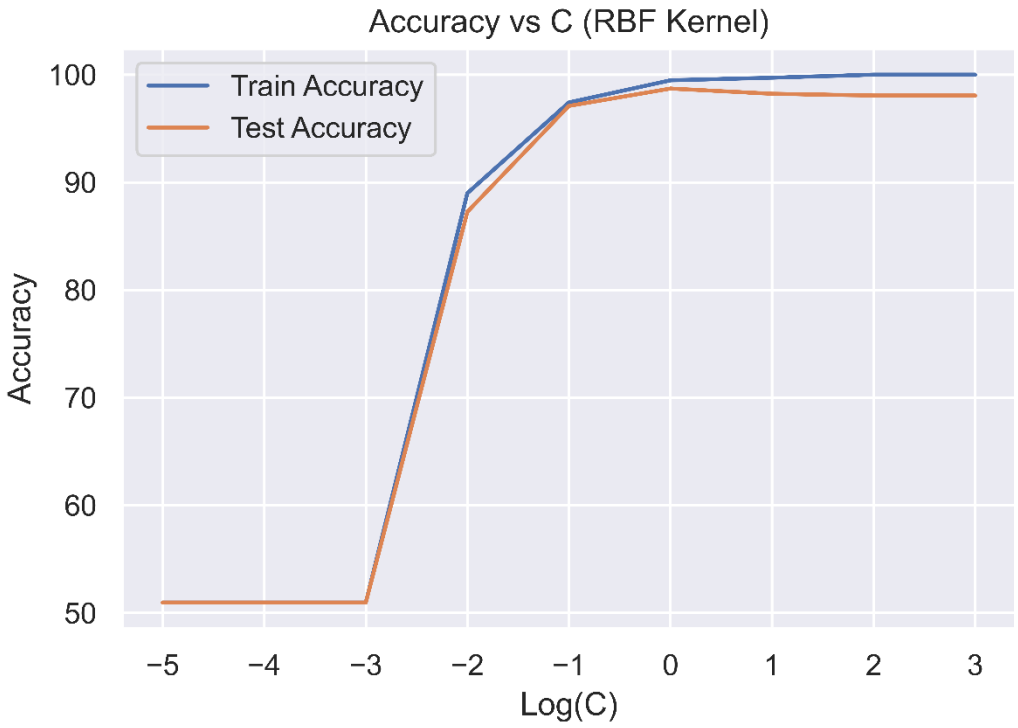



(Accuracy vs Gamma (RBF Kernel) for 25 features)

We also perform a logarithmic sweep over values of γ. We obtain the following result:

| | |
|---|---|
| $\gamma < 0.01$ | Underfitting |
| $\gamma = 0.01$ | Best fit |
| $\gamma > 0.01$ | Severe Overfitting |

Now for the case of 10 features:

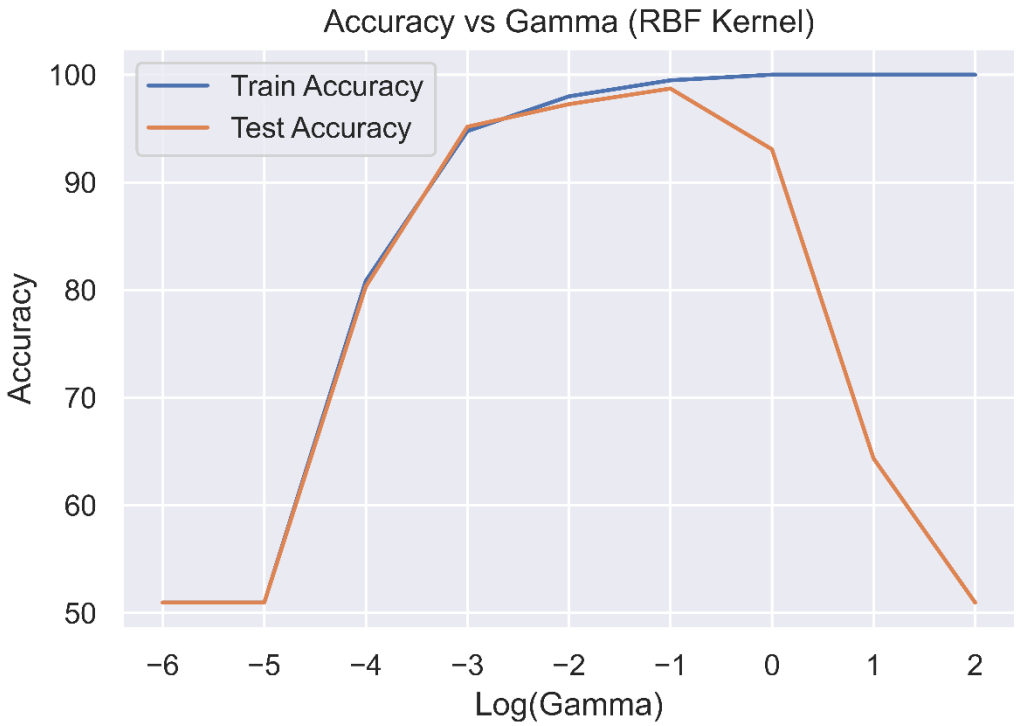



(Accuracy vs C (RBF Kernel) for 10 features)

Now, if we use only 10 features, we get a similar story, for C < 0.1, the model is underfitting the data and we get peak test accuracy at C = 1 beyond which the model starts overfitting the data.

| | |
|---|---|
| C < 1 | Underfitting |
| C = 1 | Best fit |
| C > 1 | Overfitting |

The main difference that we observe here is that the peak test accuracy is lower when we have 10 features instead of 25.




(Accuracy vs γ (RBF Kernel) for 10 features)

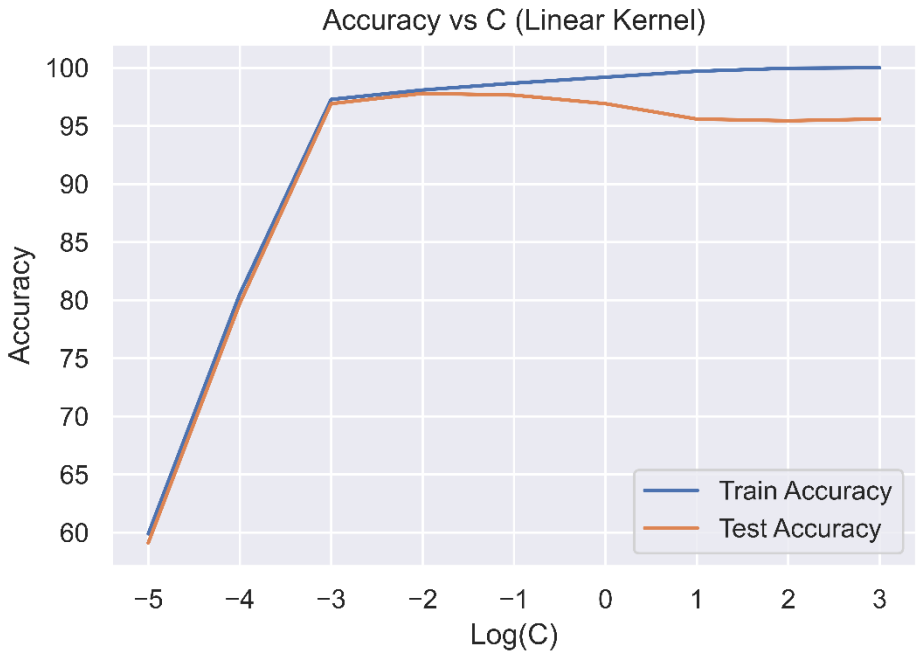
We get the following results after performing a logarithmic sweep over γ:

| | |
|---|---|
| $\gamma < 0.1$ | Underfitting |
| $\gamma = 0.1$ | Best fit |
| $\gamma > 0.1$ | Severe Overfitting |

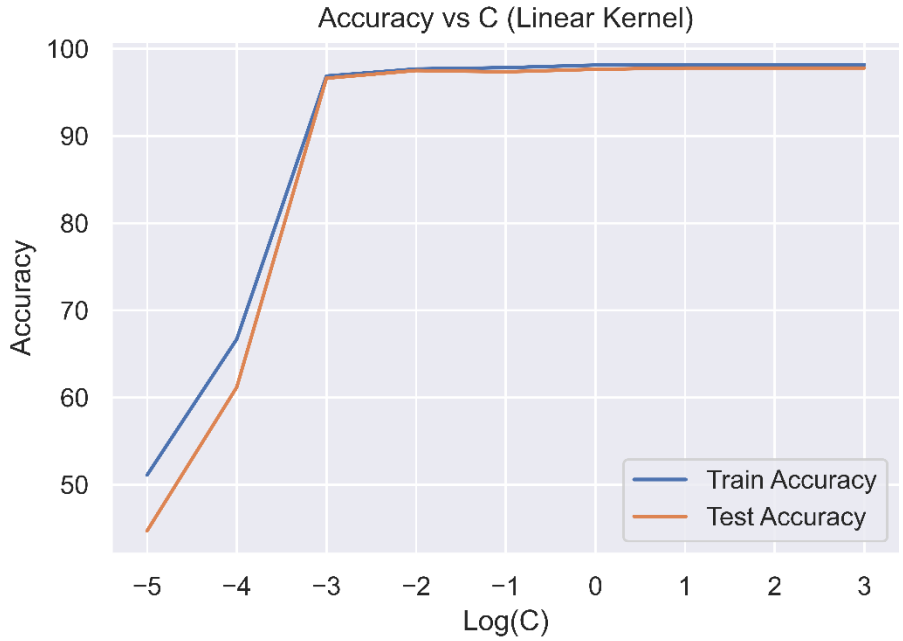
One thing to note here is that the value of γ at which peak test accuracy occurred has changed from 0.01 to 0.1, this change however is not very significant as the test accuracy is very close on both of these values.

**2.2.2** Now we do the analysis for: **C1 = 3, C2 = 7**

Using **linear** kernel:




(Accuracy vs C (Linear Kernel) for 25 features)

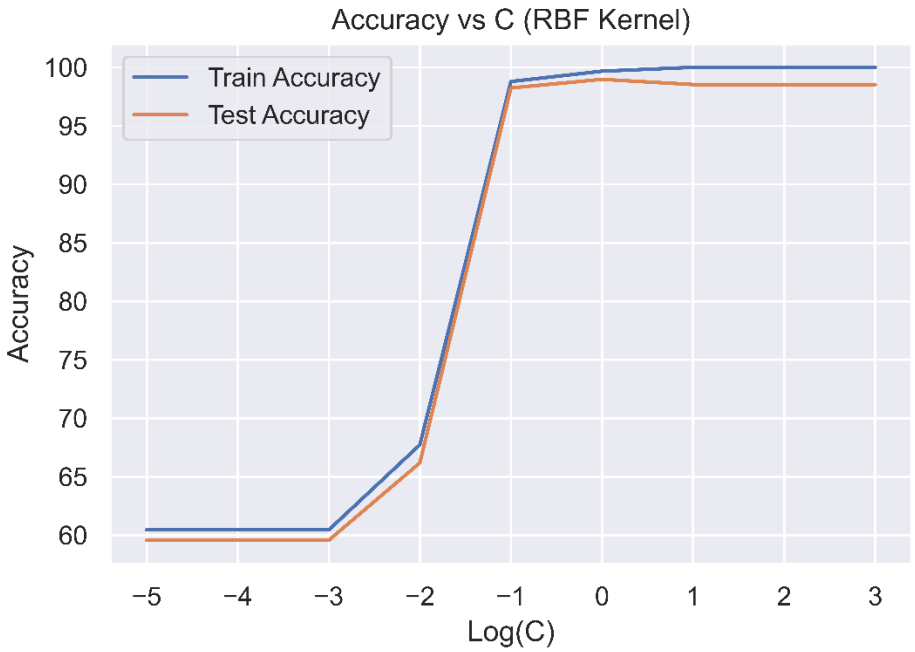



(Accuracy vs C (Linear Kernel) for 10 features)

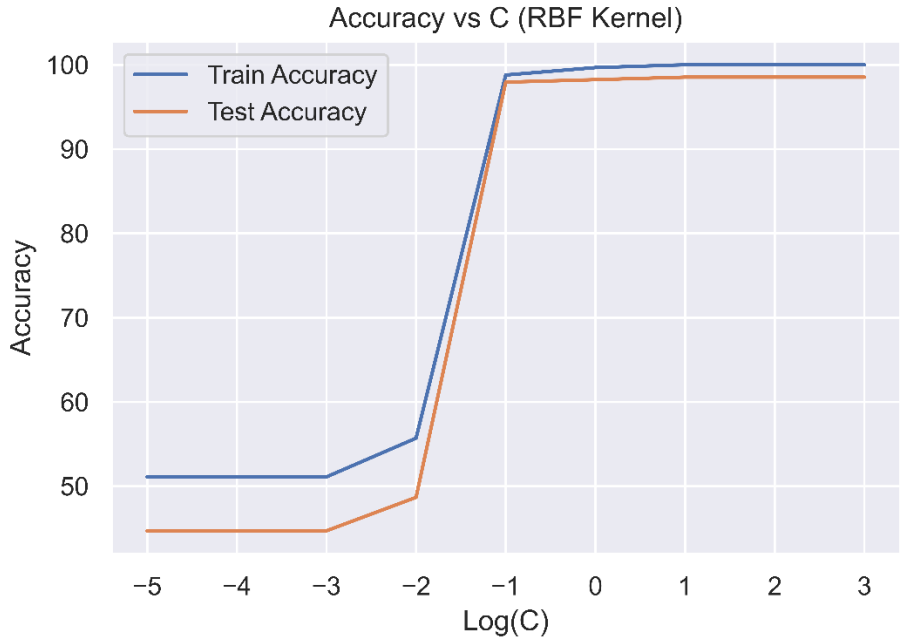
| | |
|---|---|
| C < 0.01 | Underfitting |
| C = 0.01 | Best fit |
| C > 0.01 | Overfitting |

(Consistent results)

Using **RBF** kernel:



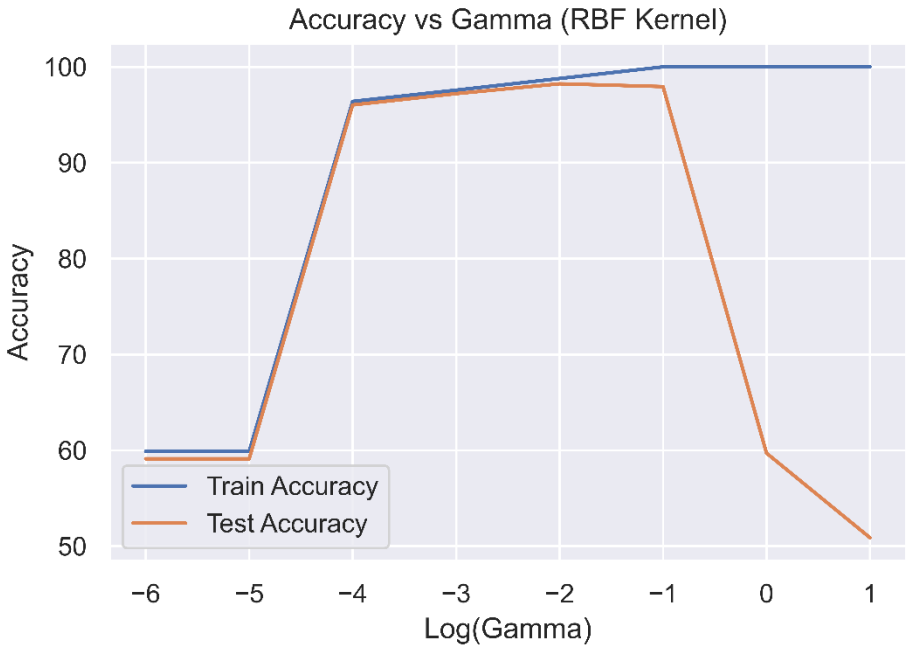

(Accuracy vs C (RBF Kernel) for 25 features)



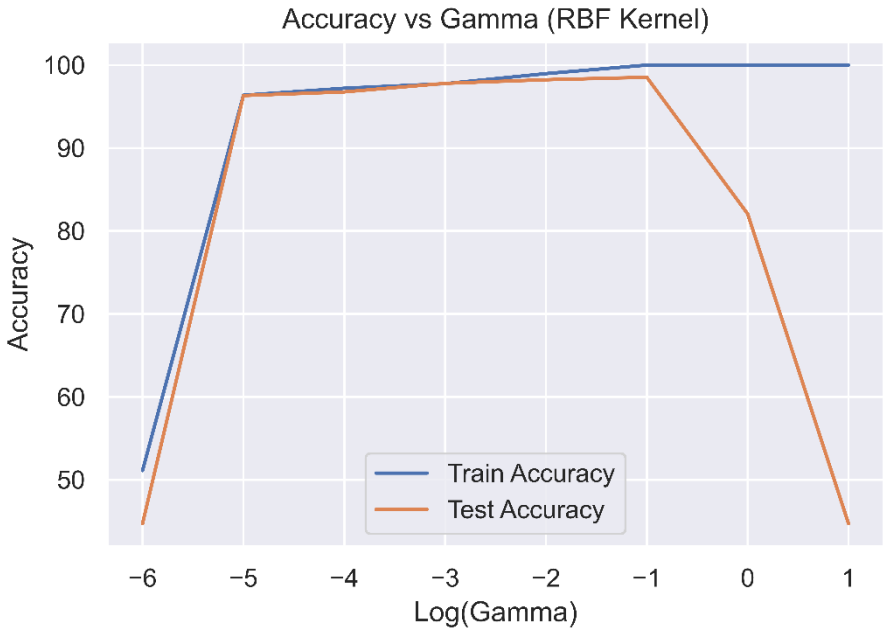

(Accuracy vs C (RBF Kernel) for 10 features)

| | |
|---|---|
| C < 0.1 | Underfitting |
| C = 0.1 | Best fit |
| C > 0.1 | Overfitting |

(Consistent results)




(Accuracy vs γ (RBF Kernel) for 25 features)



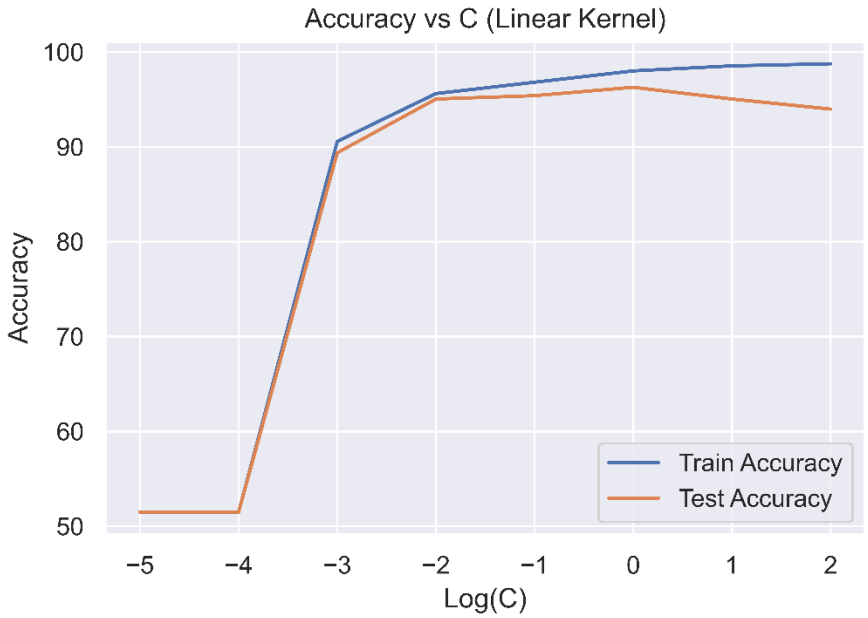

(Accuracy vs γ (RBF Kernel) for 10 features)

| | |
|---|---|
| $\gamma < 0.1$ | Underfitting |
| $\gamma = 0.1$ | Best fit |
| $\gamma > 0.1$ | Severe Overfitting |

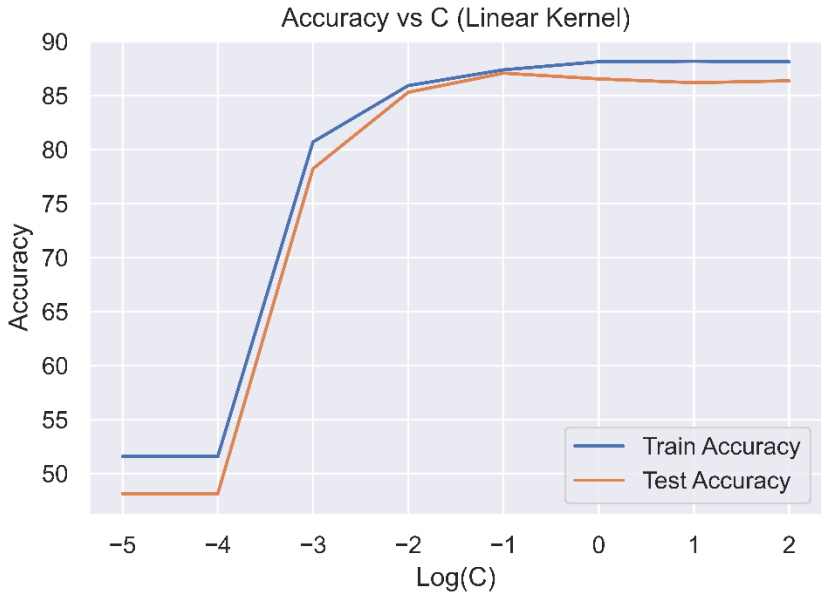
(Consistent results)

**2.2.3** Now we do the analysis for: **C1 = 4, C2 = 9**

Using **linear** kernel:



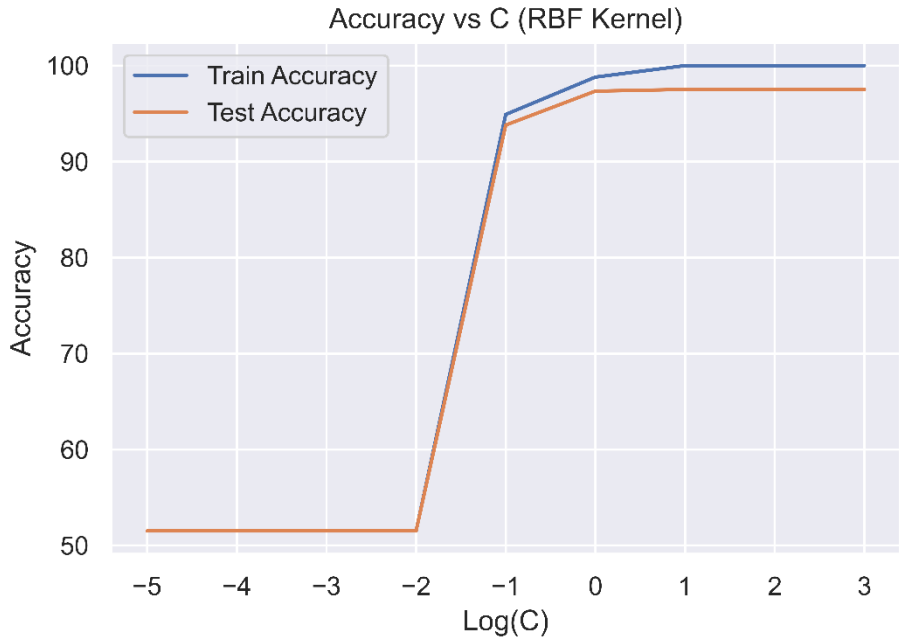

(Accuracy vs C (Linear Kernel) for 25 features)



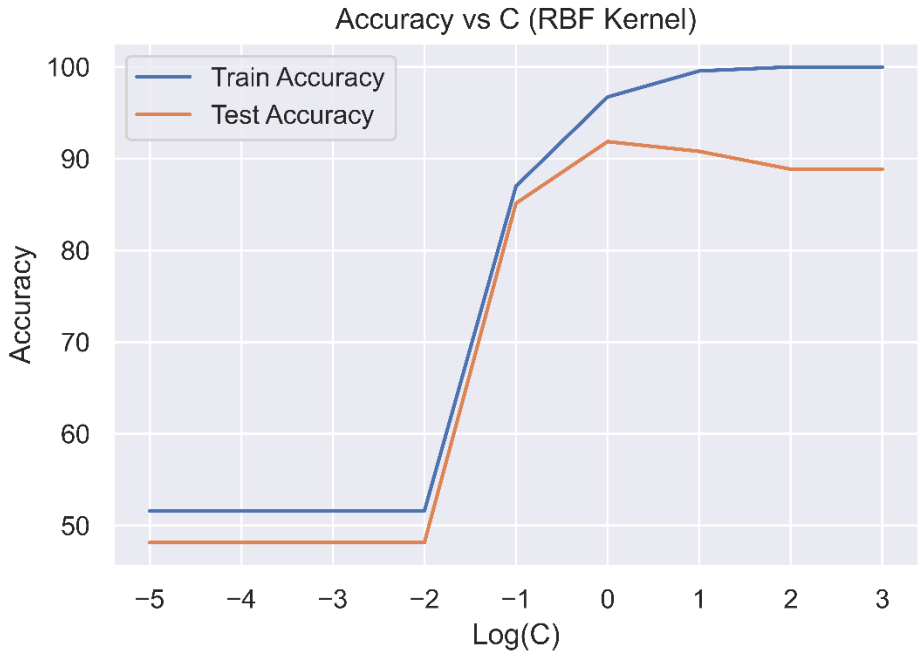

(Accuracy vs C (Linear Kernel) for 10 features)

| 25 features | 10 features | |
|---|---|---|
| C < 1 | C < 0.1 | Underfitting |
| C = 1 | C = 0.1 | Best fit |
| C > 1 | C > 0.1 | Overfitting |

Using **RBF** kernel:



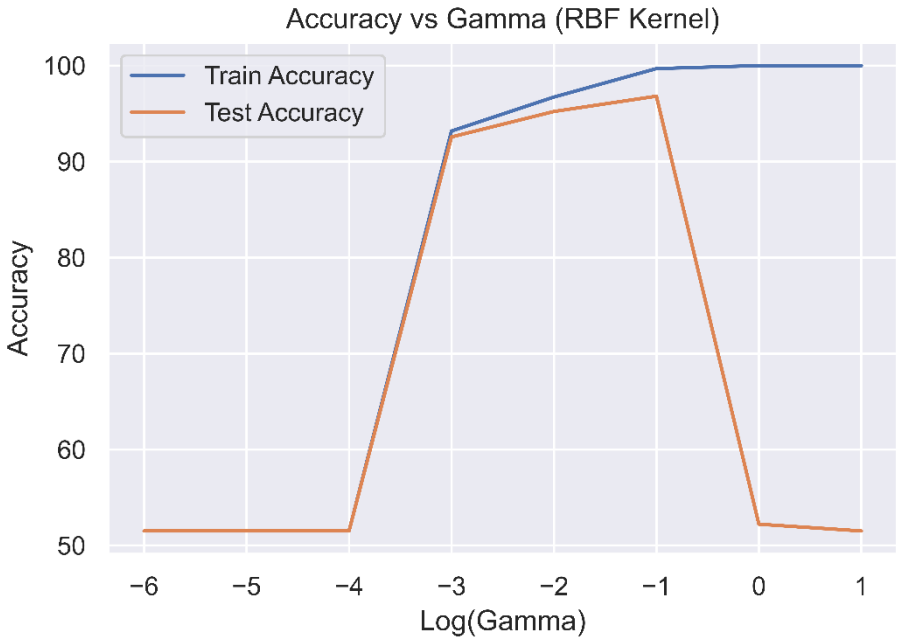

(Accuracy vs C (RBF Kernel) for 25 features)



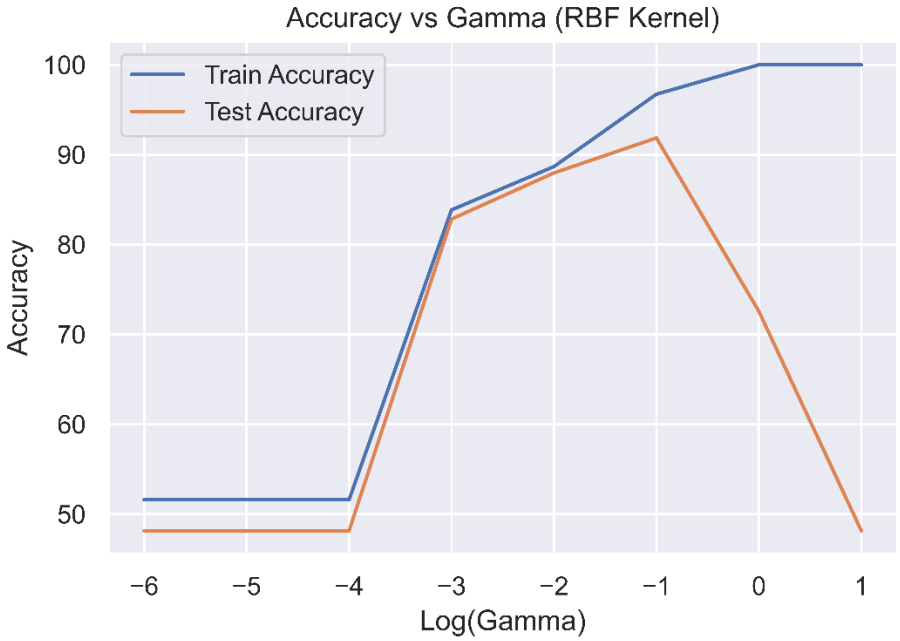

(Accuracy vs C (RBF Kernel) for 10 features)

| | |
|---|---|
| C < 1 | Underfitting |
| C = 1 | Best fit |
| C > 1 | Overfitting |

(Consistent results)




(Accuracy vs γ (RBF Kernel) for 25 features)




(Accuracy vs γ (RBF Kernel) for 10 features)

| | |
|---|---|
| $\gamma < 0.1$ | Underfitting |
| $\gamma = 0.1$ | Best fit |
| $\gamma > 0.1$ | Severe Overfitting |

(Consistent results)

### 2.2.4 Comparison of Hyperparameters

Below, I have tabulated the value of the hyperparameters at which we got the best fit while running the logarithmic sweep of the respective parameters for the three pairs of classes that we took:

| | Class C1,C2 | | |
|---|---|---|---|
| | (1,8) | (3,7) | (4,9) |
| Linear (C) | 0.1 | 0.01 | 0.1 |
| RBF (C) | 1 | 0.1 | 1 |
| RBF ($\gamma$) | 0.01 | 0.1 | 0.1 |

We observe that there are slight differences in the value of the hyperparameters at which we obtain the peak test accuracy for the three pairs of classes that we took however, the difference is not large. It's usually just a factor of 10 and despite that, the accuracies are fairly close in some cases, especially when we are considering the value of the C hyperparameter.

From the accuracies obtained, we also observe that RBF kernel performs better than linear kernel for this dataset.

## 2.3 Multiclass Classification

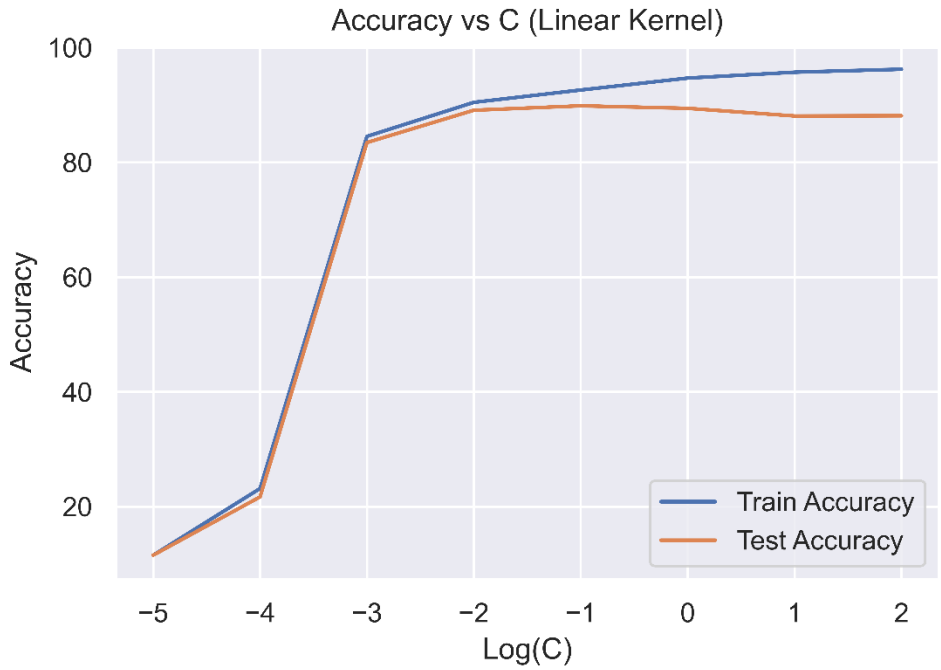
In this part of the assignment, we use the full dataset given to us (all 10 classes) and build a multiclass SVM classifier. For all the comparisons, we perform a 5-fold cross validation.

### 2.3.1 Methods used by LIBSVM library.

In my experiments, I use the LIBSVM library for python. As per its documentation given at libsvm.pdf, LIBSVM uses "one-against-one" approach for multiclass classification. Since in our dataset, we have 10 classes, then 10*9/2 = 45 classifiers are constructed and each one trains data from two classes. During the classification, the library uses a voting strategy in which each binary classification is considered to be a voting where votes can be cast for all data points x and in the end, a point is designated to that class which has the highest number of votes.

### 2.3.2 Effect of Variation in hyperparameters

First, we try out the **linear** kernel.

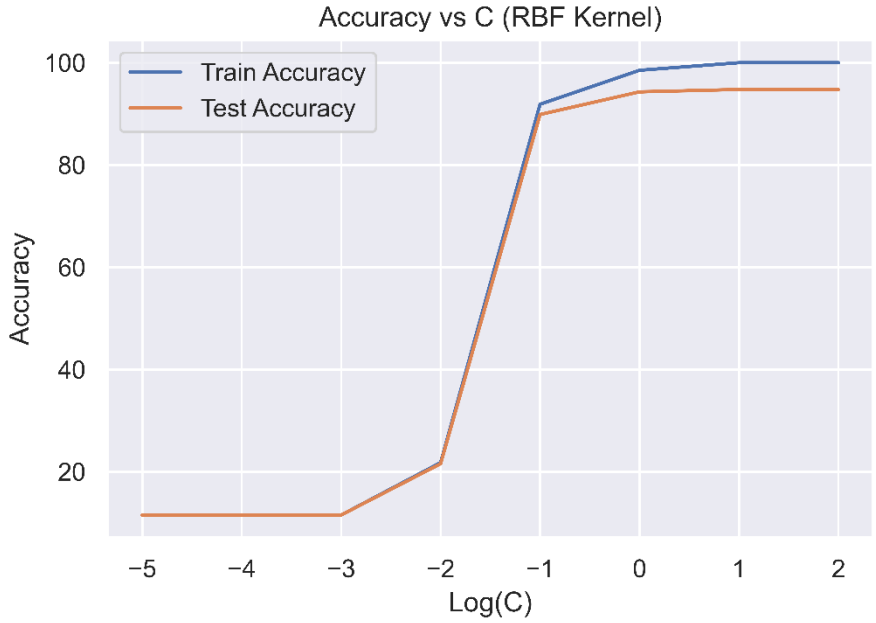



(Accuracy vs C (Linear Kernel) for 25 features)

We get the following results:

| | |
|---|---|
| C < 0.01 | Underfitting |
| C = 0.01 | Best fit |
| C > 0.01 | Overfitting |

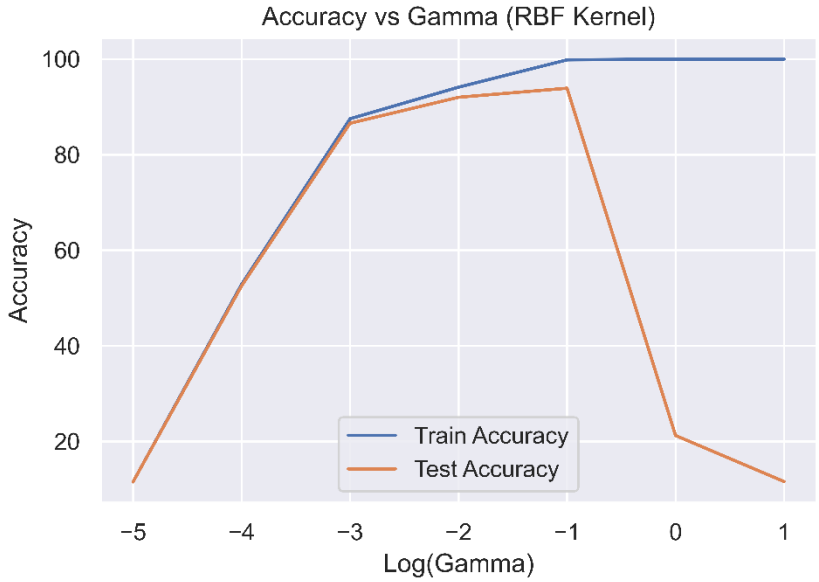
If we switch to **RBF** kernel:




(Accuracy vs C (RBF Kernel) for 25 features)

From here, we get the following results:

| | |
|---|---|
| C < 1 | Underfitting |
| C = 1 | Best fit |
| C > 1 | Overfitting |

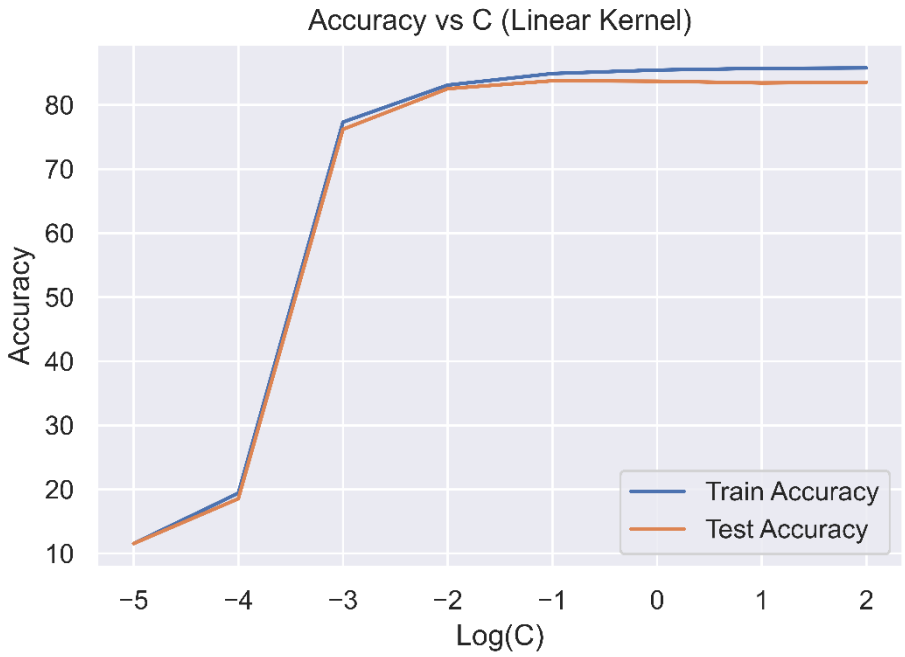



(Accuracy vs γ (RBF Kernel) for 25 features)

| | |
|---|---|
| γ < 0.1 | Underfitting |
| γ = 0.1 | Best fit |
| γ > 0.1 | Severe Overfitting |

If instead, we use only 10 features, we get the following results:

Using **linear** kernel.

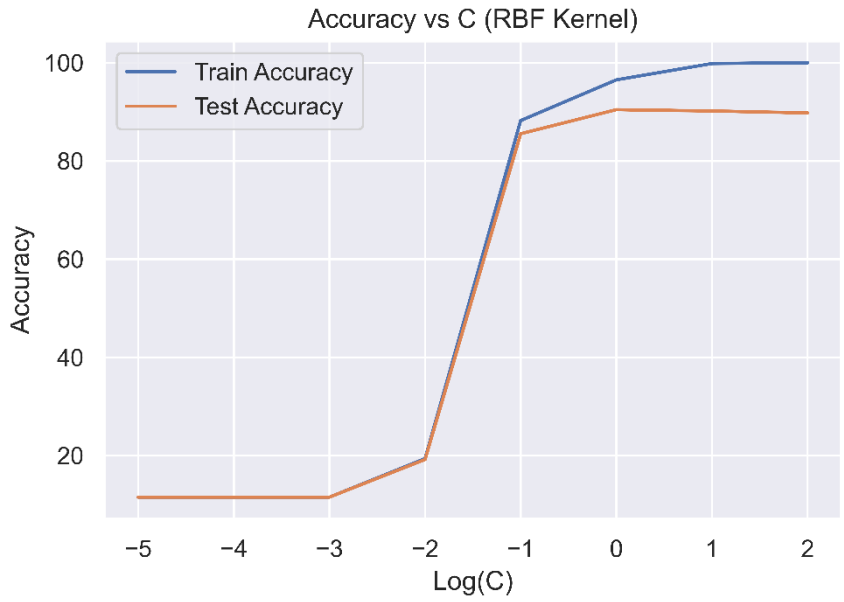



(Accuracy vs C (Linear Kernel) for 25 features)

We get the following results:

| | |
|---|---|
| C < 0.01 | Underfitting |
| C = 0.01 | Best fit |
| C > 0.01 | Overfitting |

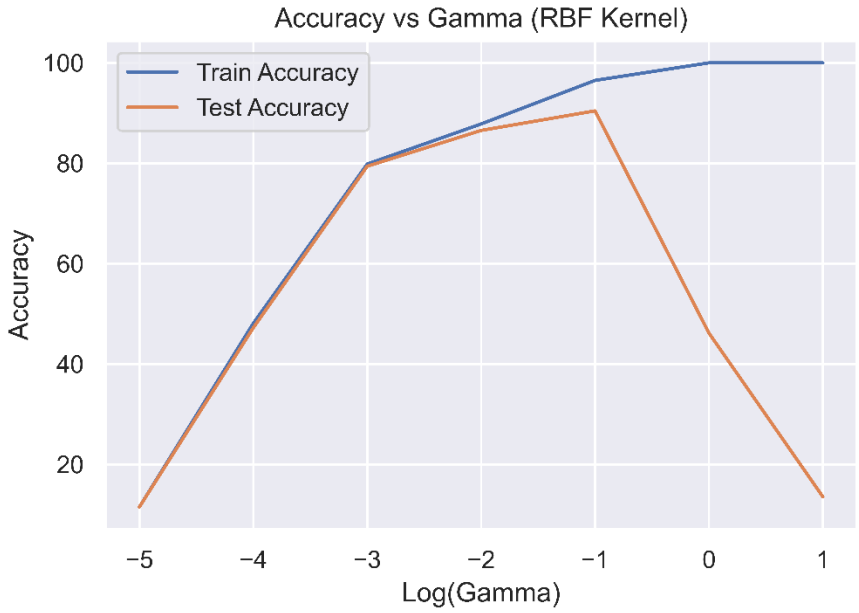
If we switch to **RBF** kernel:




(Accuracy vs C (RBF Kernel) for 25 features)

From here, we get the following results:

| | |
|---|---|
| C < 1 | Underfitting |
| C = 1 | Best fit |
| C > 1 | Overfitting |




(Accuracy vs γ (RBF Kernel) for 25 features)

| | |
|---|---|
| γ < 0.1 | Underfitting |
| γ = 0.1 | Best fit |
| γ > 0.1 | Severe Overfitting |

### 2.3.3 Fine tuning of parameters

From the above graphs for 25 features and RBF kernel, we see that we are getting the best cross validation accuracy when C lies between 0.1 and 10, and when γ lies between 0.01 and 0.1. Therefore we run a grid search for values lying this range as per the following piece of code:

```
def gridSearch():
    global C
    global GAMMA
    df = pd.DataFrame()
    bestC, bestG = 0,0
    bestAcc = 0
    for C in tqdm(np.arange(0.1,4.05,0.05)):
        for GAMMA in tqdm(np.arange(0.01,0.12,0.0025)):
            try:
                train_acc,test_acc = evaluateCrossEval(dataset_split)
                if test_acc>bestAcc:
                    bestC = C
                    bestG = GAMMA
                    bestAcc = test_acc
                it = {'C':C,'Gamma':GAMMA,'trainAcc':train_acc, 'testAcc':
↪test_acc}
                df = df.append(it,ignore_index=True)
            except:
                continue
        df.to_csv('grid_pt2.csv',index=False)
    return bestC,bestG
```

Here we have varied the parameter C from 0.1 to 4.05 with increments of 0.05, and varied γ from 0.01 to 0.12 with increments of size 0.0025. We store the test/train accuracies in a csv file and also directly return the value of value of C and γ for which we obtain the best results.

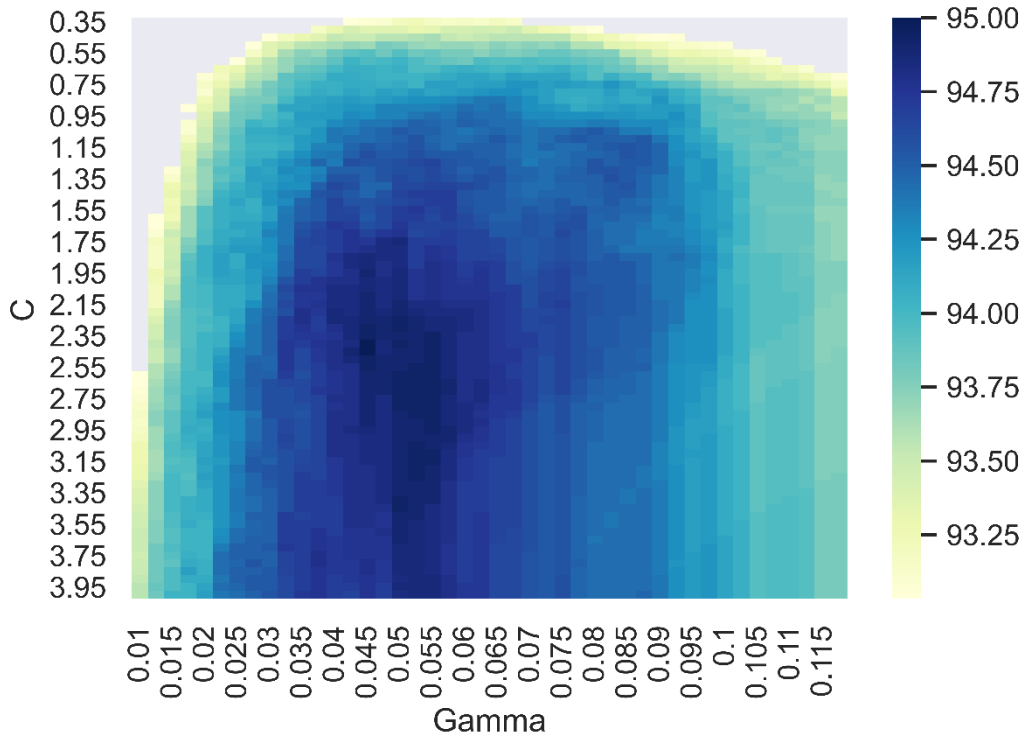
We get the best results for the following set of parameters:

| **KERNEL** | RBF |
|---|---|
| **C** | 2.4 |
| **γ** | 0.45 |

For this set of hyper-parameters, we get the following results:

| **Training Accuracy** | 99.84 % |
|---|---|
| **Testing Accuracy** | 95 % |

We also plot a heat-map for these variations and clip the test accuracies which are < 93%. Following is the heat-map:




| | Class C1,C2 | | | Multi-class |
|---|---|---|---|---|
| | (1,8) | (3,7) | (4,9) | |
| **Linear (C)** | 0.1 | 0.01 | 0.1 | 0.058 |
| **RBF (C)** | 1 | 0.1 | 1 | 2.4 |
| **RBF (γ)** | 0.01 | 0.1 | 0.1 | 0.45 |

The values obtained after fine tuning for multiclass are close enough to the values we got for the Binary classification. The differences are mostly of the order less than 10. This slight difference is because we performed a logarithmic sweep for the hyperparameters in the case of binary classification, however, in the case of multiclass classification, we fine-tuned the hyperparameters in between the powers of 10 as well.

### 2.3.4 Difference between using 10 features as compared to 25 features

The differences that we observe when using 10 features instead of 25 features are:

- The model accuracy does fall but the drop is not as significant as expected. This may be because the first 10 features in itself carry a lot of information and hence the model is able to draw separating hyperplanes from them.
- It takes lesser time to train the model when using 10 features.
- The hyper-parameters at which we get the best performance are close in the cases when we take 10 features instead of 25. At most, there's a factor of 10 difference in the value of C, but even then, the performance is very comparable at the two values.

### 2.3.5 Comparison between kernels

Finally, we also tabulate the best performances (mean of cross validation test set accuracy) that we were able to achieve from the kernels implemented:

| KERNEL | Cross Validation Acc. |
|---|---|
| Linear | 90.233 % |
| Polynomial | 92.733 % |
| RBF | 95.0 % |

As we can see, we get the best performance by using RBF kernel followed by polynomial kernel and at the end, the linear kernel.

## 3. Simplified SMO algorithm

In this part of the assignment, we implement simplified SMO algorithm for solving the optimization problem (instead of the previously implemented CVXOPT approach or the LIBSVM library).

The pseudocode for this algorithm is given at SMO.pdf and that is implemented as follows:

```
        elif self.mode == 'SSMO':
            alpha = np.zeros((num_samples))
            alpha_old = np.zeros((num_samples))
            E = np.zeros((num_samples))
            b = 0
            passes = 0
            while(passes<self.max_passes):
                num_changed_alpha = 0
                for i in range(num_samples):
                    E[i] = np.sum(alpha * y * self.K[:,i]) + b - y[i]

                    if (y[i] * E[i] < -self.tol and alpha[i]<self.C) or (y[i] *␣
↪E[i] > self.tol and alpha[i]>0):
                        j = random.randint(0,num_samples-1)
                        while j == i:
                            j = random.randint(0,num_samples-1)
```

```
                        E[j] = np.sum(alpha * y * self.K[:,j]) + b - y[j]
                        alpha_old[i] = alpha[i]
                        alpha_old[j] = alpha[j]

                        if y[i] != y[j]:
                            L = max(0,alpha[j] - alpha[i])
                            H = min(self.C,self.C+alpha[j] - alpha[i])
                        else:
                            L = max(0,alpha[i]+alpha[j] - self.C)
                            H = min(self.C,alpha[i] + alpha[j])

                        if L==H:
                            continue

                        eta = 2* self.K[i][j] - self.K[i][i] - self.K[j][j]

                        if eta>=0:
                            continue
```

```
                        eta = 2* self.K[i][j] - self.K[i][i] - self.K[j][j]

                        if eta>=0:
                            continue

                        alpha[j] = alpha[j] - y[j] * (E[i] - E[j])/eta
```

```
                        if alpha[j]>H:
                            alpha[j] = H
                        elif alpha[j]<L:
                            alpha[j] = L

                        if abs(alpha[j] - alpha_old[j]) < 1e-5:
                            continue

                        alpha[i] = alpha[i] + y[i]* y[j] * (alpha_old[j] -␣
↪alpha[j])

                        b1 = b - E[i] - y[i] * (alpha[i] - alpha_old[i]) * self.
↪K[i][i] \
                            - y[j] * (alpha[j] - alpha_old[j]) * self.K[i][j]
                        b2 = b - E[j] - y[i] * (alpha[i] - alpha_old[i]) * self.
↪K[i][j] \
                            - y[j] * (alpha[j] - alpha_old[j]) * self.K[j][j]
```

```
                        if 0<alpha[i] and alpha[i] <self.C:
                            b = b1
                        elif 0<alpha[j] and alpha[j] < self.C:
                            b = b2
                        else:
                            b = (b1+b2)/2
```

```
                        num_changed_alpha +=1

                if num_changed_alpha == 0:
                    passes+=1
                else:
                    passes=0

            self.b = b
            self.sup_idx = np.where(alpha>0)[0]
            self.sup_x = X[self.sup_idx,:]
            self.sup_y = y[self.sup_idx]
            self.alpha = alpha[self.sup_idx]
            self.w = np.dot(alpha*y,X).T
```

Now we compare the performance of the Simplified SMO algorithm with the previously implemented approaches. For this, we will be using a **5-fold cross validation** approach with the same parameters and class pairs that we got for the binary classification section of the assignment:

| | Class C1,C2 | | |
|---|---|---|---|
| | (1,8) | (3,7) | (4,9) |
| Linear (C) | 0.1 | 0.01 | 0.1 |
| RBF (C) | 1 | 0.1 | 1 |
| RBF (γ) | 0.01 | 0.1 | 0.1 |

## V. C1 = 1, C2 = 8 (Linear Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 98.0241935483871 | 97.25806451612902 | 00.160021 |
| CVXOPT | 97.94354838709677 | 97.09677419354838 | 06.822120 |
| S-SMO | 97.8225806451613 | 97.25806451612902 | 13.569374 |

## VI. C1 = 1, C2 = 8 (RBF Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 98.30645161290322 | 98.06451612903224 | 00.171265 |
| CVXOPT | 98.46774193548387 | 98.06451612903224 | 14.654499 |
| S-SMO | 98.26612903225806 | 98.2258064516129 | 21.069174 |

## VII. C1 = 4, C2 = 9 (Linear Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 97.0796460176991 | 96.10619469026547 | 00.150144 |
| CVXOPT | 96.99115044247787 | 95.75221238938052 | 06.162015 |
| S-SMO | 97.03539823008848 | 96.10619469026548 | 17.039943 |

## VIII. C1 = 4, C2 = 9 (RBF Kernel)

| | Training Acc. (%) | Testing Acc. (%) | Time Taken (s) |
|---|---|---|---|
| LIBSVM | 99.64601769911505 | 97.34513274336283 | 00.219928 |
| CVXOPT | 99.69026548672568 | 97.16814159292035 | 19.763364 |
| S-SMO | 99.64601769911505 | 97.34513274336283 | 20.779926 |

From the experiments above, we can deduce the following:

- The performance of the models are very similar when trained using the three implementations (LIBSVM, CVXOPT, and S-SMO).
- The time taken by the implementations varies as: **LIBSVM << CVXOPT < S-SMO**. This is because of the very fast and optimized implementation being used in LIBSVM (full SMO).

Next, we also compare the support vectors that we get from the three approaches.

## V. C1 = 1, C2 = 8 (Linear Kernel)

```
Mode: LIBSVM
Training Accuracy: 97.63948497854076
Testing Accuracy: 97.43589743589743
63 number of support vectors found
The indices of the support vectors are: [5, 11, 13, 15, 22, 31, 36, 40, 44, 46, 56, 60, 63, 76, 78, 80, 86, 91, 111, 115, 118, 1
38, 139, 143, 148, 151, 153, 161, 168, 172, 182, 210, 221, 236, 242, 245, 246, 259, 281, 293, 300, 305, 307, 310, 317, 322, 339,
342, 343, 345, 351, 356, 359, 360, 367, 372, 394, 395, 398, 399, 401, 417, 462]

Mode: CVXOPT
Training Accuracy: 97.63948497854076
Testing Accuracy: 97.43589743589743
63 number of support vectors found
The indices of the support vectors are: [5, 11, 13, 15, 22, 31, 36, 40, 44, 46, 56, 60, 63, 76, 78, 80, 86, 91, 111, 115, 118, 1
38, 139, 143, 148, 151, 153, 161, 168, 172, 182, 210, 221, 236, 242, 245, 246, 259, 281, 293, 300, 305, 307, 310, 317, 322, 339,
342, 343, 345, 351, 356, 359, 360, 367, 372, 394, 395, 398, 399, 401, 417, 462]

Mode: SSMO
Training Accuracy: 97.63948497854076
Testing Accuracy: 97.43589743589743
65 number of support vectors found
The indices of the support vectors are: [5, 11, 13, 15, 22, 31, 36, 40, 44, 46, 56, 60, 63, 76, 78, 80, 86, 91, 111, 115, 118, 1
25, 138, 139, 143, 148, 151, 153, 161, 168, 172, 182, 210, 221, 236, 242, 245, 246, 259, 281, 289, 293, 300, 305, 307, 310, 317,
322, 339, 342, 343, 345, 351, 356, 359, 360, 367, 372, 394, 395, 398, 399, 401, 417, 462]
```

## VI. C1 = 1, C2 = 8 (RBF Kernel)

```
Mode: LIBSVM
Training Accuracy: 98.49785407725322
Testing Accuracy: 96.15384615384616
38 number of support vectors found
The indices of the support vectors are: [5, 13, 17, 21, 31, 36, 46, 56, 60, 78, 80, 86, 91, 111, 115, 118, 119, 143, 148, 161, 1
68, 182, 221, 245, 259, 281, 317, 339, 342, 343, 345, 351, 352, 359, 372, 395, 399, 401]

Mode: CVXOPT
Training Accuracy: 98.49785407725322
Testing Accuracy: 96.15384615384616
38 number of support vectors found
The indices of the support vectors are: [5, 13, 17, 21, 31, 36, 46, 56, 60, 78, 80, 86, 91, 111, 115, 118, 119, 143, 148, 161, 1
68, 182, 221, 245, 259, 281, 317, 339, 342, 343, 345, 351, 352, 359, 372, 395, 399, 401]

Mode: SSMO
Training Accuracy: 98.49785407725322
Testing Accuracy: 96.15384615384616
39 number of support vectors found
The indices of the support vectors are: [5, 13, 17, 21, 31, 36, 46, 56, 60, 78, 80, 86, 91, 111, 115, 118, 119, 143, 148, 161, 1
68, 182, 221, 245, 259, 281, 317, 339, 342, 343, 345, 351, 352, 356, 359, 372, 395, 399, 401]
```

## VII. C1 = 4, C2 = 9 (Linear Kernel)

```
Mode: LIBSVM
Training Accuracy: 96.93396226415094
Testing Accuracy: 96.47887323943662
89 number of support vectors found
The indices of the support vectors are: [2, 3, 5, 8, 12, 13, 14, 15, 16, 18, 21, 24, 26, 27, 30, 33, 36, 50, 56, 58, 64, 73, 74,
80, 81, 83, 93, 103, 104, 110, 123, 125, 136, 137, 144, 148, 157, 166, 167, 168, 170, 173, 174, 175, 177, 178, 179, 188, 196, 19
7, 205, 214, 219, 232, 233, 242, 244, 246, 255, 257, 261, 262, 266, 272, 275, 289, 293, 295, 306, 307, 318, 334, 336, 341, 344,
349, 358, 360, 364, 370, 382, 387, 389, 390, 394, 396, 406, 409, 410]

Mode: CVXOPT
Training Accuracy: 97.16981132075472
Testing Accuracy: 97.1830985915493
89 number of support vectors found
The indices of the support vectors are: [2, 3, 5, 8, 12, 13, 14, 15, 16, 18, 21, 24, 26, 27, 30, 33, 36, 50, 56, 58, 64, 73, 74,
80, 81, 83, 93, 103, 104, 110, 123, 125, 136, 137, 144, 148, 157, 166, 167, 168, 170, 173, 174, 175, 177, 178, 179, 188, 196, 19
7, 205, 214, 219, 232, 233, 242, 244, 246, 255, 257, 261, 262, 266, 272, 275, 289, 293, 295, 306, 307, 318, 334, 336, 341, 344,
349, 358, 360, 364, 370, 382, 387, 389, 390, 394, 396, 406, 409, 410]

Mode: SSMO
Training Accuracy: 96.93396226415094
Testing Accuracy: 96.47887323943662
90 number of support vectors found
The indices of the support vectors are: [2, 3, 5, 8, 12, 13, 14, 15, 16, 18, 21, 24, 26, 27, 30, 33, 36, 50, 56, 58, 64, 73, 74,
80, 81, 83, 93, 103, 104, 110, 123, 125, 136, 137, 144, 148, 157, 166, 167, 168, 170, 173, 174, 175, 177, 178, 179, 188, 196, 19
7, 205, 214, 219, 232, 233, 242, 244, 246, 255, 257, 261, 262, 266, 272, 275, 289, 293, 295, 306, 307, 311, 318, 334, 336, 341,
344, 349, 358, 360, 364, 370, 382, 387, 389, 390, 394, 396, 406, 409, 410]
```

## VIII. C1 = 4, C2 = 9 (RBF Kernel)

```
Mode: LIBSVM
Training Accuracy: 99.76415094339622
Testing Accuracy: 97.1830985915493
319 number of support vectors found
The indices of the support vectors are: [0, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 35, 36, 38, 39, 40, 42, 43, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65,
66, 67, 69, 70, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 98, 100, 101, 103, 104,
105, 107, 109, 110, 112, 114, 115, 116, 117, 119, 120, 121, 123, 124, 125, 126, 127, 129, 130, 131, 133, 134, 136, 138, 141, 14
3, 144, 145, 147, 148, 149, 151, 152, 153, 154, 155, 156, 157, 159, 161, 163, 164, 165, 166, 167, 168, 170, 172, 173, 174, 175,
177, 178, 179, 180, 182, 183, 185, 186, 187, 188, 189, 193, 194, 195, 196, 197, 198, 201, 202, 205, 206, 208, 210, 211, 212, 21
3, 214, 215, 217, 219, 220, 221, 223, 225, 226, 227, 228, 230, 232, 233, 235, 236, 237, 238, 240, 242, 244, 246, 251, 252, 253,
255, 256, 257, 258, 259, 261, 262, 263, 266, 267, 268, 270, 272, 273, 274, 275, 276, 278, 281, 282, 283, 284, 287, 289, 290, 29
2, 293, 294, 295, 297, 298, 299, 300, 302, 304, 305, 306, 307, 310, 311, 312, 313, 315, 318, 319, 320, 321, 322, 323, 326, 327,
328, 329, 330, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 35
5, 356, 358, 359, 360, 361, 362, 363, 364, 365, 367, 368, 369, 370, 372, 374, 375, 376, 377, 378, 379, 381, 382, 384, 387, 388,
389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 402, 403, 405, 406, 407, 408, 409, 410, 412, 413, 414, 416, 420, 42
1, 422, 423]

Mode: CVXOPT
Training Accuracy: 99.76415094339622
Testing Accuracy: 97.1830985915493
330 number of support vectors found
The indices of the support vectors are: [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 35, 36, 38, 39, 40, 42, 43, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64,
65, 66, 67, 69, 70, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 98, 100, 101, 103, 1
04, 105, 107, 109, 110, 112, 114, 115, 116, 117, 119, 120, 121, 123, 124, 125, 126, 127, 129, 130, 131, 133, 134, 136, 137, 138,
139, 141, 143, 144, 145, 147, 148, 149, 151, 152, 153, 154, 155, 156, 157, 159, 161, 162, 163, 164, 165, 166, 167, 168, 170, 17
2, 173, 174, 175, 177, 178, 179, 180, 182, 183, 185, 186, 187, 188, 189, 193, 194, 195, 196, 197, 198, 201, 202, 205, 206, 208,
210, 211, 212, 213, 214, 215, 217, 219, 220, 221, 223, 225, 226, 227, 228, 230, 232, 233, 235, 236, 237, 238, 240, 242, 244, 24
6, 251, 252, 253, 255, 256, 257, 258, 259, 261, 262, 263, 266, 267, 268, 269, 270, 272, 273, 274, 275, 276, 278, 279, 281, 282,
283, 284, 287, 289, 290, 292, 293, 294, 295, 297, 298, 299, 300, 302, 304, 305, 306, 307, 310, 311, 312, 313, 315, 318, 319, 32
0, 321, 322, 323, 326, 327, 328, 329, 330, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349,
350, 351, 352, 353, 354, 355, 356, 358, 359, 360, 361, 362, 363, 364, 365, 367, 368, 369, 370, 371, 372, 374, 375, 376, 377, 37
8, 379, 380, 381, 382, 384, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 402, 403, 405, 406, 407,
408, 409, 410, 412, 413, 414, 416, 417, 419, 420, 421, 422, 423]
```

```
Mode: SSMO
Training Accuracy: 99.76415094339622
Testing Accuracy: 97.1830985915493
319 number of support vectors found
The indices of the support vectors are: [0, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 35, 36, 38, 39, 40, 42, 43, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65,
66, 67, 69, 70, 72, 73, 74, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 95, 96, 98, 100, 101, 103, 104,
105, 107, 109, 110, 112, 114, 115, 116, 117, 119, 120, 121, 123, 124, 125, 126, 127, 129, 130, 131, 133, 134, 136, 138, 141, 14
3, 144, 145, 147, 148, 149, 151, 152, 153, 154, 155, 156, 157, 159, 161, 163, 164, 165, 166, 167, 168, 170, 172, 173, 174, 175,
177, 178, 179, 180, 182, 183, 185, 186, 187, 188, 189, 193, 194, 195, 196, 197, 198, 201, 202, 205, 206, 208, 210, 211, 212, 21
3, 214, 215, 217, 219, 220, 221, 223, 225, 226, 227, 228, 230, 232, 233, 235, 236, 237, 238, 240, 242, 244, 246, 251, 252, 253,
255, 256, 257, 258, 259, 261, 262, 263, 266, 267, 268, 270, 272, 273, 274, 275, 276, 278, 281, 282, 283, 284, 287, 289, 290, 29
2, 293, 294, 295, 297, 298, 299, 300, 302, 304, 305, 306, 307, 310, 311, 312, 313, 315, 318, 319, 320, 321, 322, 323, 326, 327,
328, 329, 330, 332, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 35
5, 356, 358, 359, 360, 361, 362, 363, 364, 365, 367, 368, 369, 370, 372, 374, 375, 376, 377, 378, 379, 381, 382, 384, 387, 388,
389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 402, 403, 405, 406, 407, 408, 409, 410, 412, 413, 414, 416, 420, 42
1, 422, 423]
```

We observe that using Simplified SMO we get mostly the same support vectors although, sometimes SSMO captures an extra supporting vector. This very slight difference was expected since we are not using the full-fledged SMO algorithm.

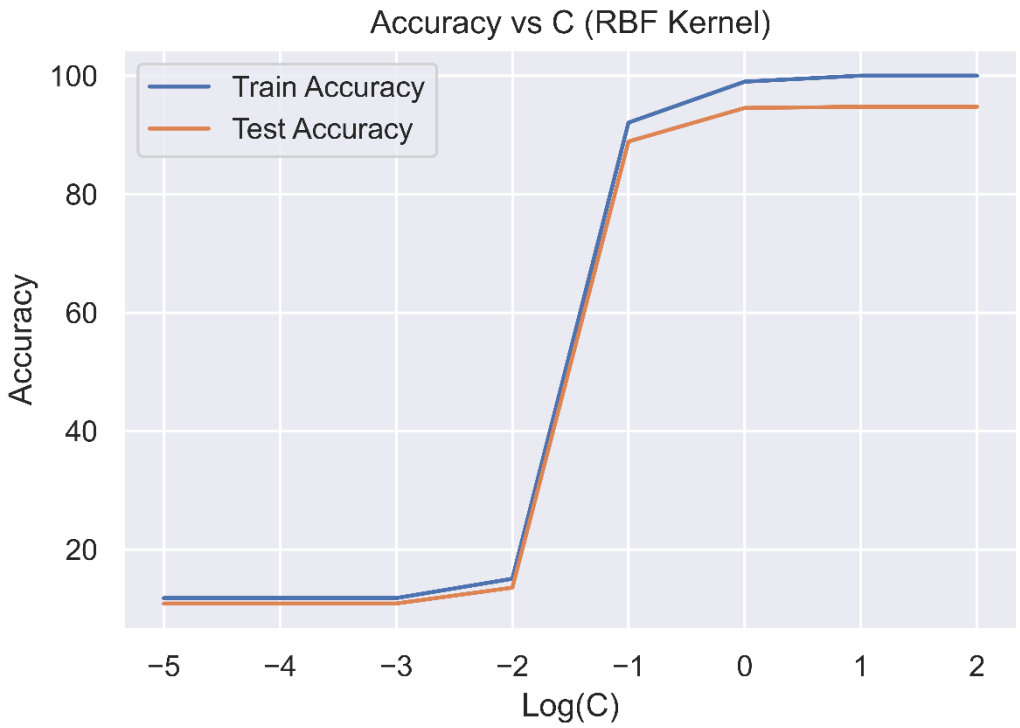
## 4. Part 2 (Kaggle competition)

In this part of the assignment, we are given a training dataset similar to the previous parts. One difference is, this time we are given a training data consisting of 8000 data points.

The complexity of LIBSVM in fitting to a dataset varies as $O(n^3)$, hence, we can't use the whole dataset for tuning the parameters as that will take a very long time. Therefore, we make a change in our usual cross validation approach. Instead of using a (k-1):1 split for training: testing, we use 1: (k-1) split. For illustration, if we have a 4 fold cross validation, we will train on 2000 data points and use the rest of 6000 points for testing in our first fold.

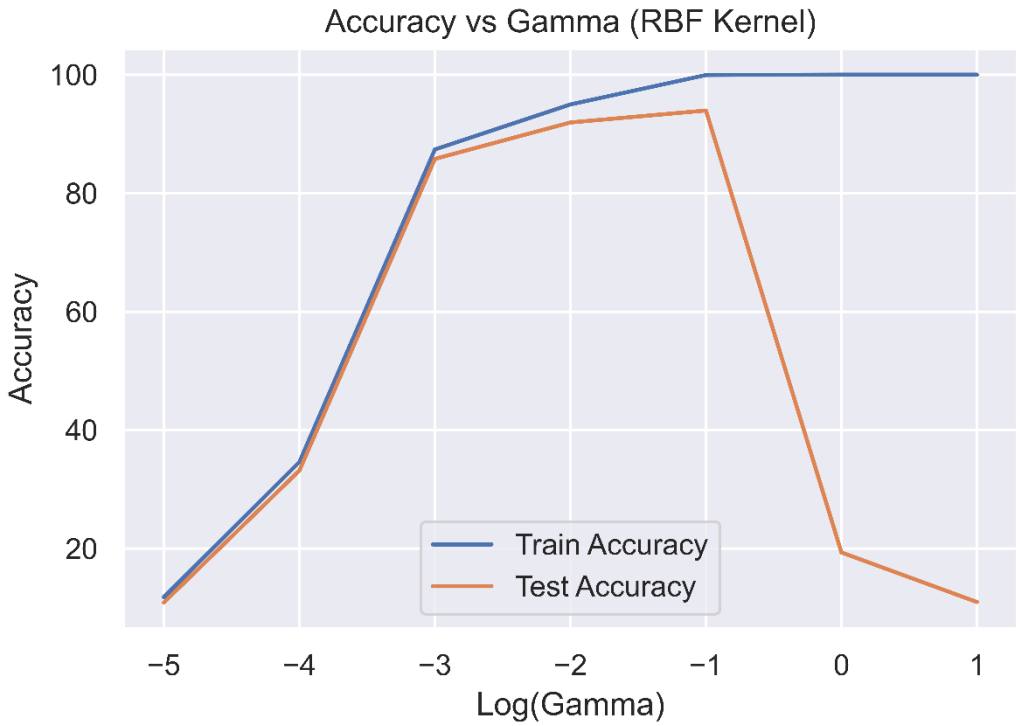
For all our experiments, we use a **5-fold cross validation** and use the **RBF kernel** as that seemed to perform the best from our previous experiments.

We plot the following variations to get an idea of where to perform the grid search.




(Accuracy vs C plot)

Here we have fixed the value of $\gamma = \frac{1}{Number\ of\ features}$ as is usually done by the standard libraries. From this plot, we can deduce that the value of C at which maxima will occur will lie between 0.1 and 10 after which the model starts overfitting.




(Accuracy vs γ plot)

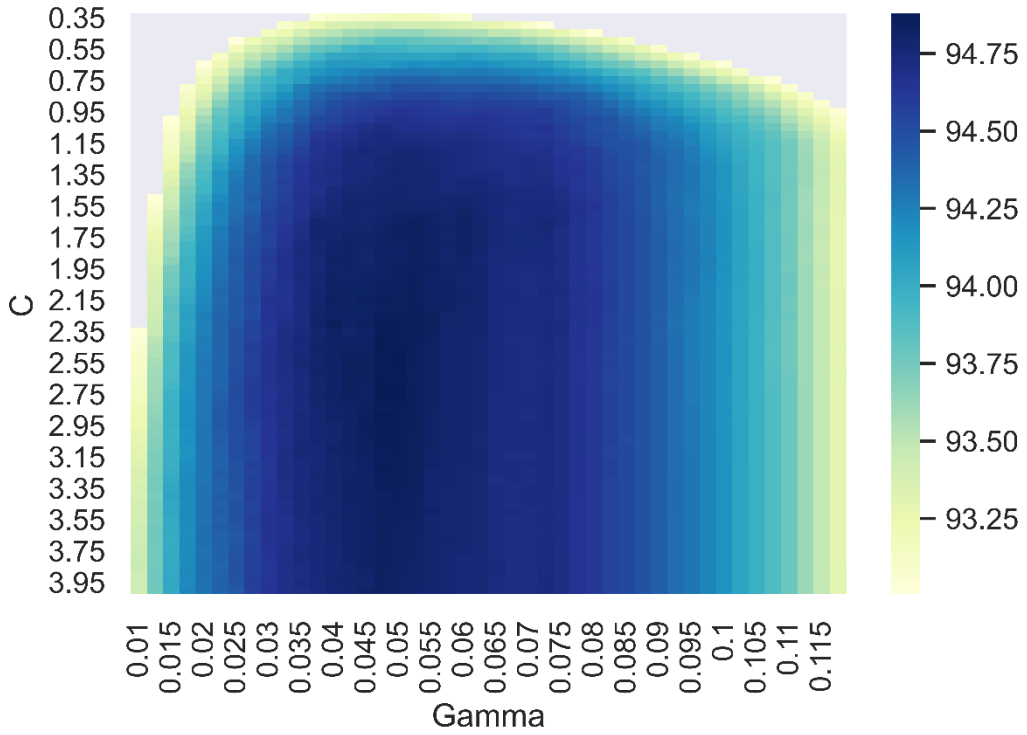
In this experiment, we have fixed the value of C to be 1 and performed a logarithmic sweep on the values of γ. From the plot, we can see that the best value of γ will lie between 0.01 and 0.1.

From the above range of values decided, we perform a grid search over these parameters and store the test/train accuracies. Following is the code snippet we follow for this:

```
def gridSearch():
    global C
    global GAMMA
    df = pd.DataFrame()
    bestC, bestG = 0,0
    bestAcc = 0
    for C in tqdm(np.arange(0.1,4.05,0.05)):
        for GAMMA in tqdm(np.arange(0.01,0.12,0.0025)):
            try:
                train_acc,test_acc = evaluateCrossEval(dataset_split)
                if test_acc>bestAcc:
                    bestC = C
                    bestG = GAMMA
                    bestAcc = test_acc
                it = {'C':C,'Gamma':GAMMA,'trainAcc':train_acc, 'testAcc':
↪test_acc}
                df = df.append(it,ignore_index=True)
            except:
                continue
        df.to_csv('grid_pt2.csv',index=False)
    return bestC,bestG
```

Here we have varied the parameter C from 0.1 to 4.05 with increments of 0.05, and varied γ from 0.01 to 0.12 with increments of size 0.0025.

Next, we plot a heat-map for these variations and clip the test accuracies which are < 93%. Following is the heat-map:

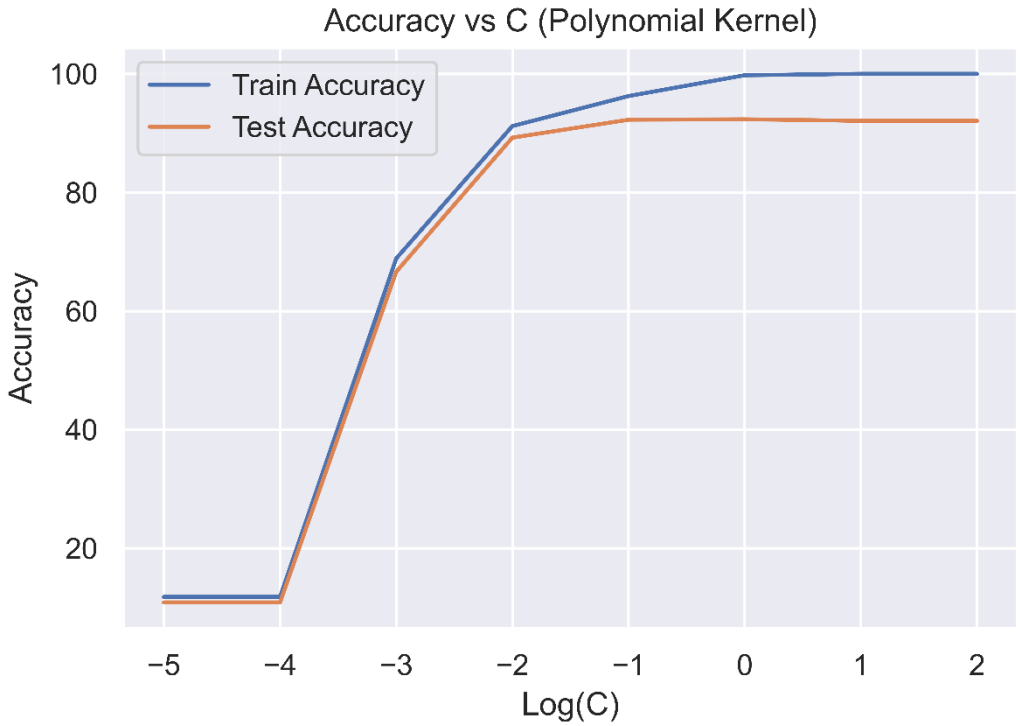



**Moreover, we get the best results for:**

**C = 2.5 and γ = 0.05**

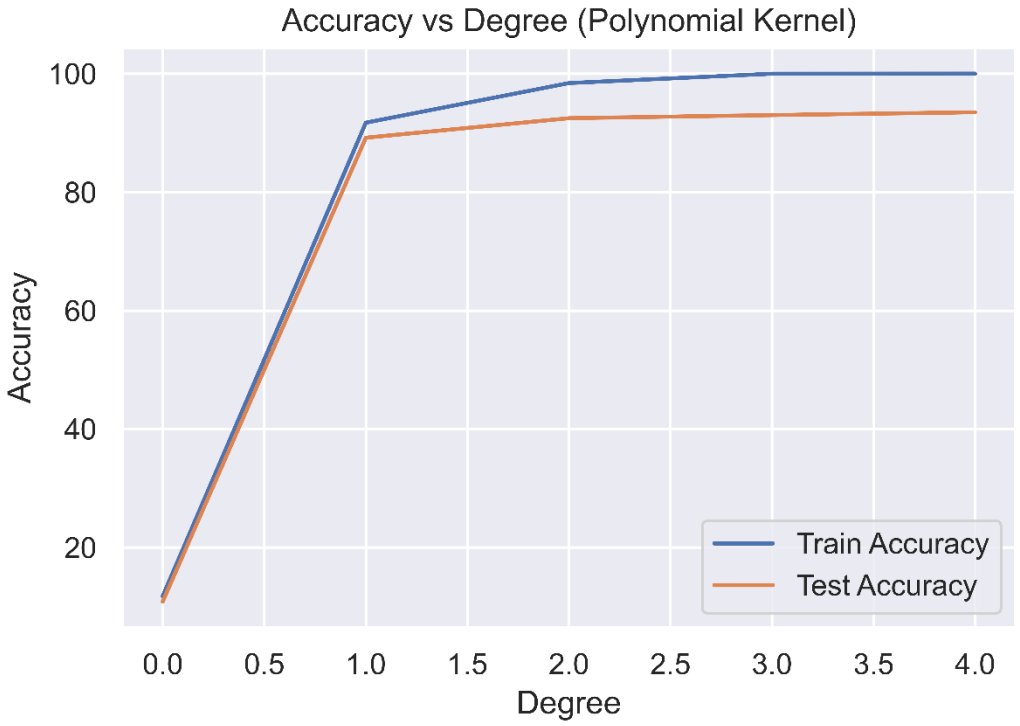
For this hyperparameter setting, we then train the model on the full 8000 point dataset and generate predictions for the test set. We get a categorization **accuracy score of 0.97** on the submission made on Kaggle.

I also tried using the **POLYNOMIAL KERNEL**, for the Kaggle task and the following were the analysis:



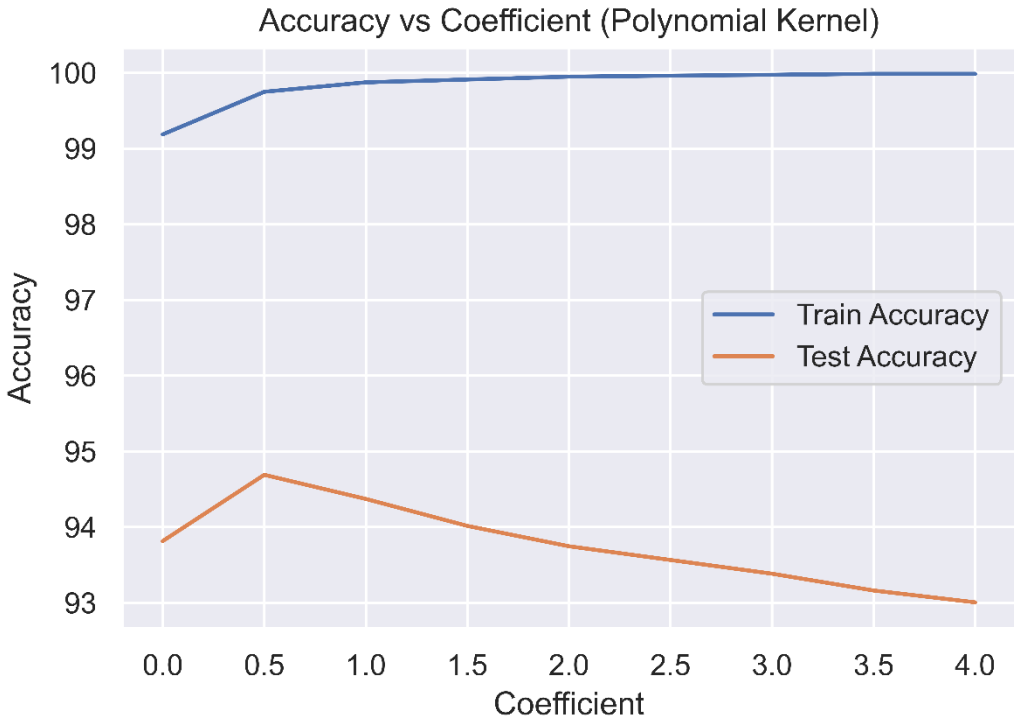

(Accuracy vs C - Polynomial Kernel)

Here we fix the value of DEGREE = 2 and COEFF = 4 and perform a logarithmic sweep of the values of C. We see that the best accuracy occurs when C is between 0.01 and 1.




(Accuracy vs Degree - Polynomial Kernel)

Here, we fix the value of C at 0.3 and perform a sweep for DEGREE. Best value of degree will lie between 1 and 4.




(Accuracy vs Coefficient - Polynomial Kernel)

Here, we fix the value of degree to be 3 and perform a sweep over values of Coefficient and we observe a peak around 0.5. Best values between 0 and 2.

Finally, we perform another grid search and we get the best value of parameters as follows:

C = 0.26

DEGREE = 3

COEFF = 0.5

For these set of parameters, we get a score of **0.96625 on Kaggle**, slightly lower than the RBF kernel.

## References


- https://courses.csail.mit.edu/6.867/wiki/images/a/a7/Qp-cvxopt.pdf
- https://stats.stackexchange.com/questions/451868/calculating-the-value-of-b-in-an-svm
- https://tullo.ch/articles/svm-py/
- https://web.iitd.ac.in/~sumeet/smo.pdf
- https://web.iitd.ac.in/~sumeet/tr-98-14.pdf
- https://www.youtube.com/watch?v=vqoVIchkM7I