Section 2

Required files: kernel.py

$K(x, x') =$

- Linear Kernel

$$(x')^T x$$

- Polynomial Kernel

$$(\zeta + \gamma(x')^T x)^Q$$

- Radial Basis Function Kernel

$$e^{-\gamma\left||x-x'|\right|^2}$$

- Sigmoid Kernel

$$\tanh\left(\gamma(x')^T x + r\right)$$

- Laplacian Kernel

$$e^{-\gamma\left||x-x'|\right|_1}$$

Section 3.1

Required files: SVM_binary.py

fit() function is divided into the following parts

- Input from csv
- Calculation of kernel
- Solving dual of soft-margin SVM using qpsolver for α

Soft margin SVM is formulated as

$$\min_{\alpha} \frac{1}{2} \sum_{m=1}^{N} \sum_{n=1}^{N} y_n y_m \alpha_n \alpha_m \boldsymbol{x}_n^T \boldsymbol{x}_m - \sum_{n=1}^{N} \alpha_n$$

$$\text{s.t.} \sum_{n=1}^{N} y_n \alpha_n = 0$$

$$0 \leq \alpha_n \leq C \ \forall n$$

Source: COL341 Homework 2

- Storing indices of support vectors
- Calculating and storing *b*

predict() function is divided into the following parts

- Calculation of g(x) without implicit calculation of *w*
- Label

Addition of threshold to improve performance

I have modified the existing clause of $\alpha > 0\ and\ \alpha < C$ to $\alpha > thres\ and\ \alpha < C$.

This was done to reduce the number of required support vectors and improve the run-time performance of the code. Care was taken to ensure this would not considerably impact the value of w by comparing the overall range of Kernel values (i.e., there was no such case where Kernel value would be incredibly large to compensate for the small value of alpha). This $thres$ was however, determined empirically and may not perform well for other training sets.

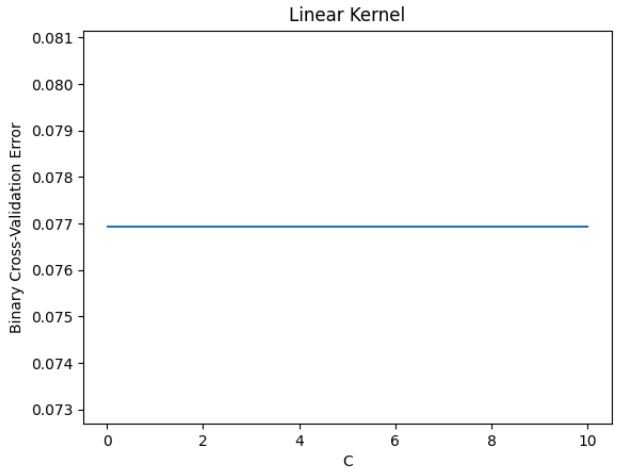
Using QP Solver: *ecos*

Section 3.2

Required files: analysis.py

- Linear Kernel Confusion Matrices (Same for all C)

		Actual	
		1	-1
Predicted	1	16	2
	-1	0	21



No observable change with C in linear kernel

- RBF Kernel Confusion Matrices
- C = {0.1,0.01} Any Γ

		Actual	
		1	-1
Predicted	1	0	21
	-1	18	0

All labels are -1

- C = 1 Γ = 0.1

		Actual	
		1	-1
Predicted	1	16	2
	-1	2	19

- C = 1 Γ = 0.01

		Actual	
		1	-1
Predicted	1	12	0
	-1	6	21

- C = 1 Γ = 0.001

		Actual	
		1	-1
Predicted	1	13	0
	-1	5	21

- C = 10 Γ = 0.1

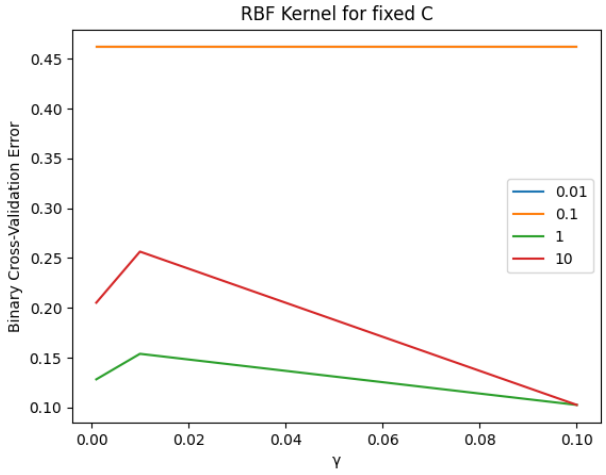
		Actual	
		1	-1
Predicted	1	16	2
	-1	2	19

- C = 10 Γ = 0.01

		Actual	
		1	-1
Predicted	1	8	0
	-1	10	21

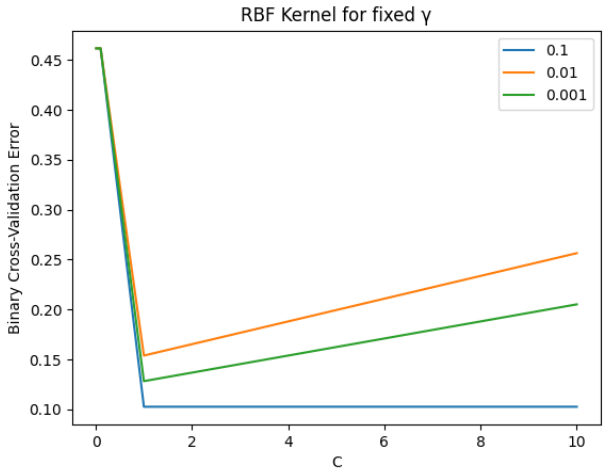
- C = 10 Γ = 0.001

		Actual	
		1	-1
Predicted	1	10	0
	-1	8	21



(*Note:* Curves for C = 0.01 and C = 0.1 overlap)

Exponential function e^x increases slowly for small x, and rapidly for large x. Hence, small values of γ are detrimental since they do not provide sufficient change in values after exponentiation (i.e., values group together for smaller γ, and classifications worsens).



For very small values of C, slack variables can arbitrarily grow large, thus allowing misclassification to easily occur. For very large values of C, slack variables diminish, and the soft margin SVM tends to a hard margin SVM, which can lead to misclassification for non-linearly separable data.

Best performance is for linear kernel, independent of C ($\mathrm{E}_{\mathrm{cv}} = 0.07692307692307693$).

We will choose linear kernel with C = 1 in competitive part (using better performance over rbf as parameter for selection of C).

Section 4

Required files: SVM_multiclass.py

In One-Vs-Rest, we make use of *N* binary classifiers, corresponding to each of the N labels. As the name suggest, we classify as label A or label ~A. We then compare the value across each of the N classifiers for a given *x,* and label it as the one with the greatest value.

In One-Vs-One, we make use of $\frac{N(N-1)}{2}$ binary classifiers, corresponding to each 2-label pair. We then count the most repeated label and set it for the given *x*. Ties between any two labels are resolved by calling the specific Trainer using both these labels, and selecting the one it outputs.

Error metric for multi-class classification is taken as average of sum of absolute deviation from expected value.

Number of states = 10 (Labels 1 through 10)

(*Note:* Code is built on the assumption that labels are from 1 to N_{states}, if not a simple bijective mapping is sufficient to bring it to this state)

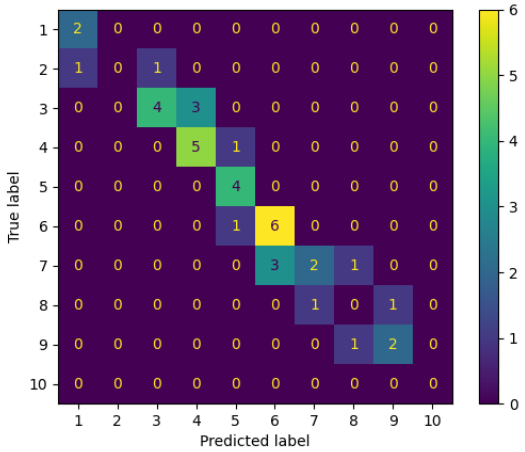
	OVA	OVO
C = 1	0.358974	1.051282
C = 0.1	5.666667	4.923077

Error

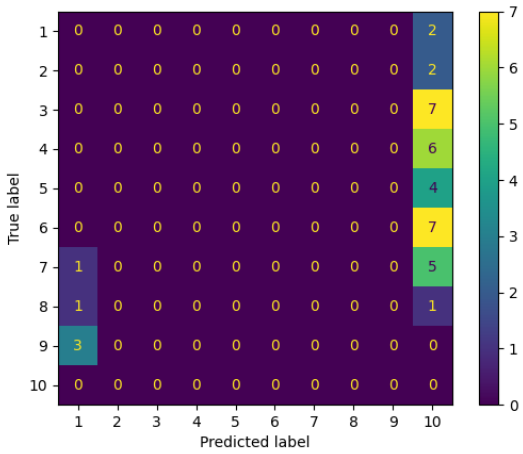
Similar argument to section 3.2 holds for the error shown above, C = 1 is close to the optimal C, and C = 0.1 is too small and this leads to misclassification.

Both C = 0.1 cases are extremely errored, with OVA classifying all as 0 or 10, and OVO classifying all as 10 (what it classifies as is implementation dependent, and hence biased. I have followed the normal counting hierarchy in all lists).

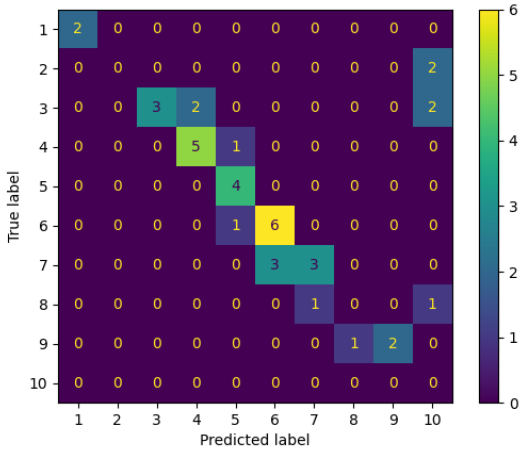
Confusion matrices (Using *matplotlib*):



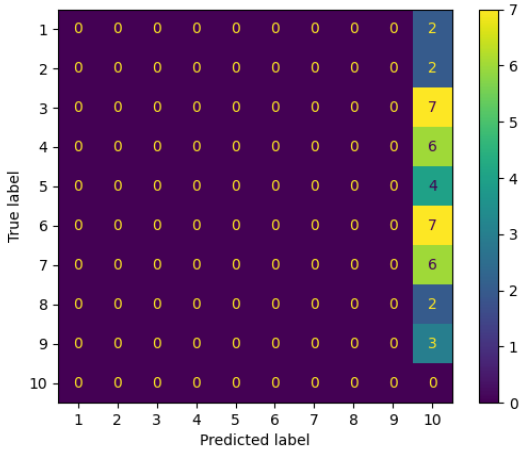
C = 1 OVA



C = 0.1 OVA



C = 1 OVO



C = 0.1 OVO

OVA performs better than OVO, and will be used in the competitive part with C = 1.

Section 5

Required files: best.py

- For Two-class,

Kernel = Linear

C = 1

- For Multi-class,

One-Versus-All

Kernel = rbf

C = 1

Γ = 0.1

Extra

QP Solvers

For all quadratic programming solvers, we have made use of *ecos*. However other solvers may perform better on our given dataset. On a loose observation, bi_train.csv is dense, and may perform better on dense algorithm solvers (*Ex*: cvxopt, proxqp, qpoases, quadprog and nppro(License-locked)). However, since none of these are included in the base installation of *qpsolvers*, use of it has been avoided.