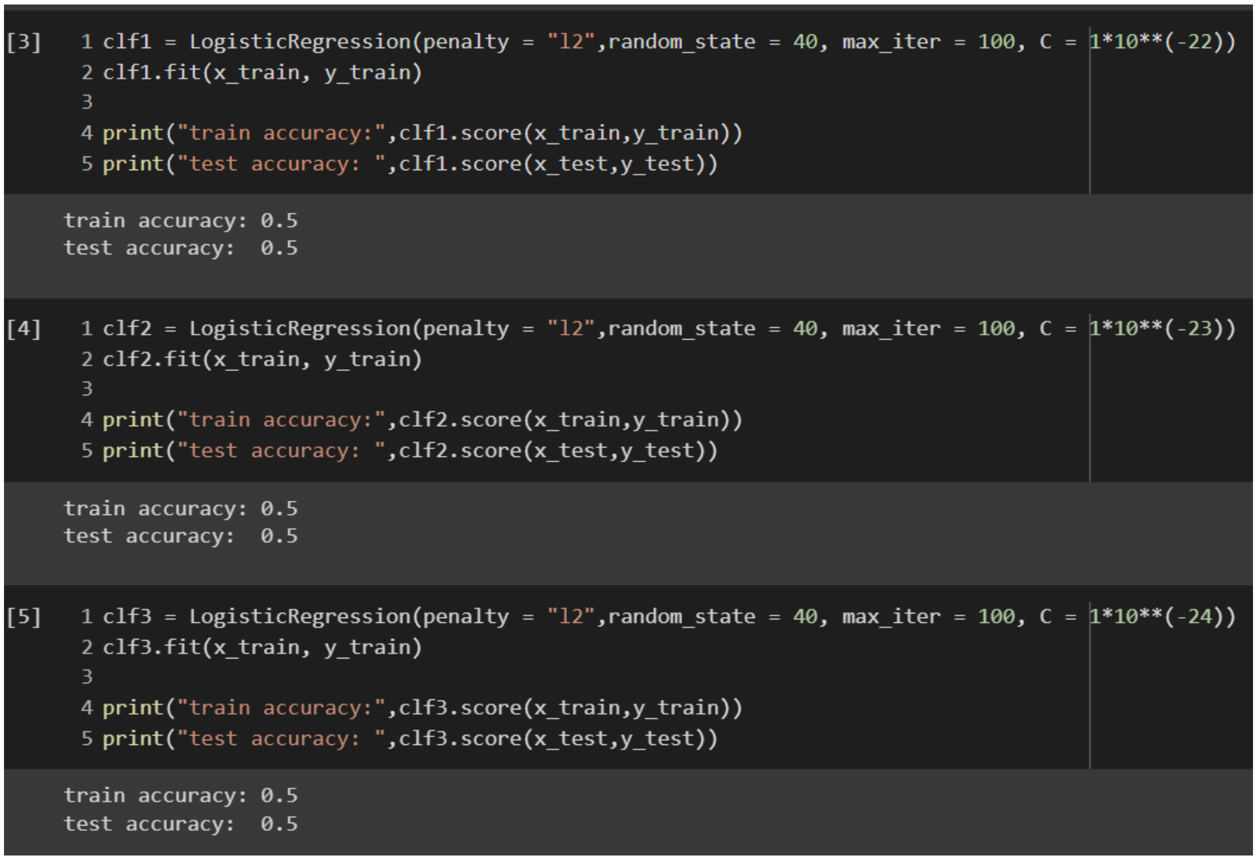
- Loaded the dataset using Keras and used only the data points (from both the training and test sets) from classes 0 and 6 (corresponding to “T-shirt/top” and “Shirt” respectively).
- To make training easier, normalized each example through rescaling the inputs by dividing them by 255, so they lie in the range [0, 1]
- In order to complete the assignment within a reasonable amount of time reduced the size of the original training set of each class to 1000 each.
- Note that if ‘x’ or ‘y’ = Reduced training set however, if ‘X’ or ‘Y’ = Full training set

First : Logistic regression



```
[3]
clf1 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 100, C = 1*10**(-22))
clf1.fit(x_train, y_train)

print("train accuracy:",clf1.score(x_train,y_train))
print("test accuracy: ",clf1.score(x_test,y_test))
```

```
train accuracy: 0.5
test accuracy:  0.5
```

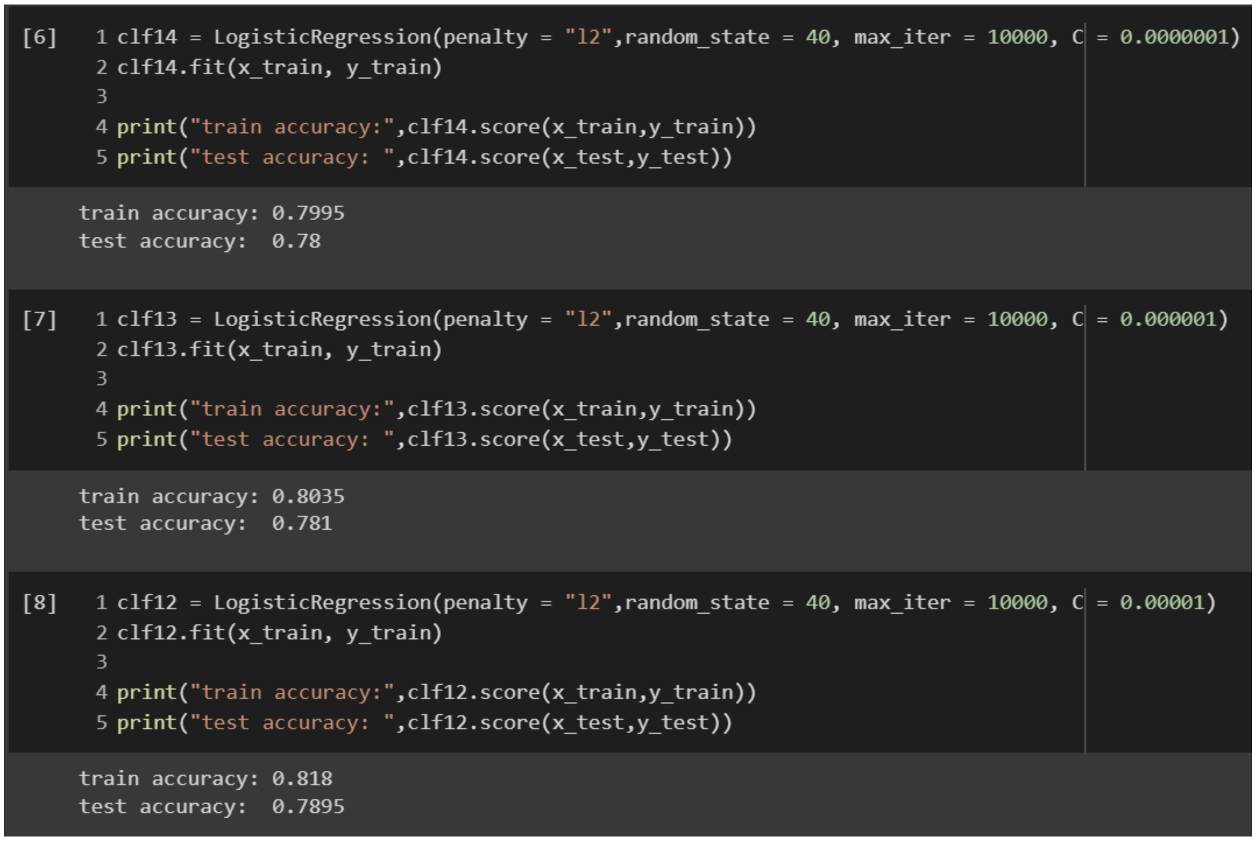
```
[4]
clf2 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 100, C = 1*10**(-23))
clf2.fit(x_train, y_train)

print("train accuracy:",clf2.score(x_train,y_train))
print("test accuracy: ",clf2.score(x_test,y_test))
```

```
train accuracy: 0.5
test accuracy:  0.5
```

```
[5]
clf3 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 100, C = 1*10**(-24))
clf3.fit(x_train, y_train)

print("train accuracy:",clf3.score(x_train,y_train))
print("test accuracy: ",clf3.score(x_test,y_test))
```

```
train accuracy: 0.5
test accuracy:  0.5
```

When c is very low both train and test accuracy stay at 0.5. Even though there is no over or underfitting the accuracy level of both the test and training set is very low.



[6]

```python
clf14 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 10000, C = 0.0000001)
clf14.fit(x_train, y_train)

print("train accuracy:",clf14.score(x_train,y_train))
print("test accuracy: ",clf14.score(x_test,y_test))
```

```
train accuracy: 0.7995
test accuracy:  0.78
```

[7]

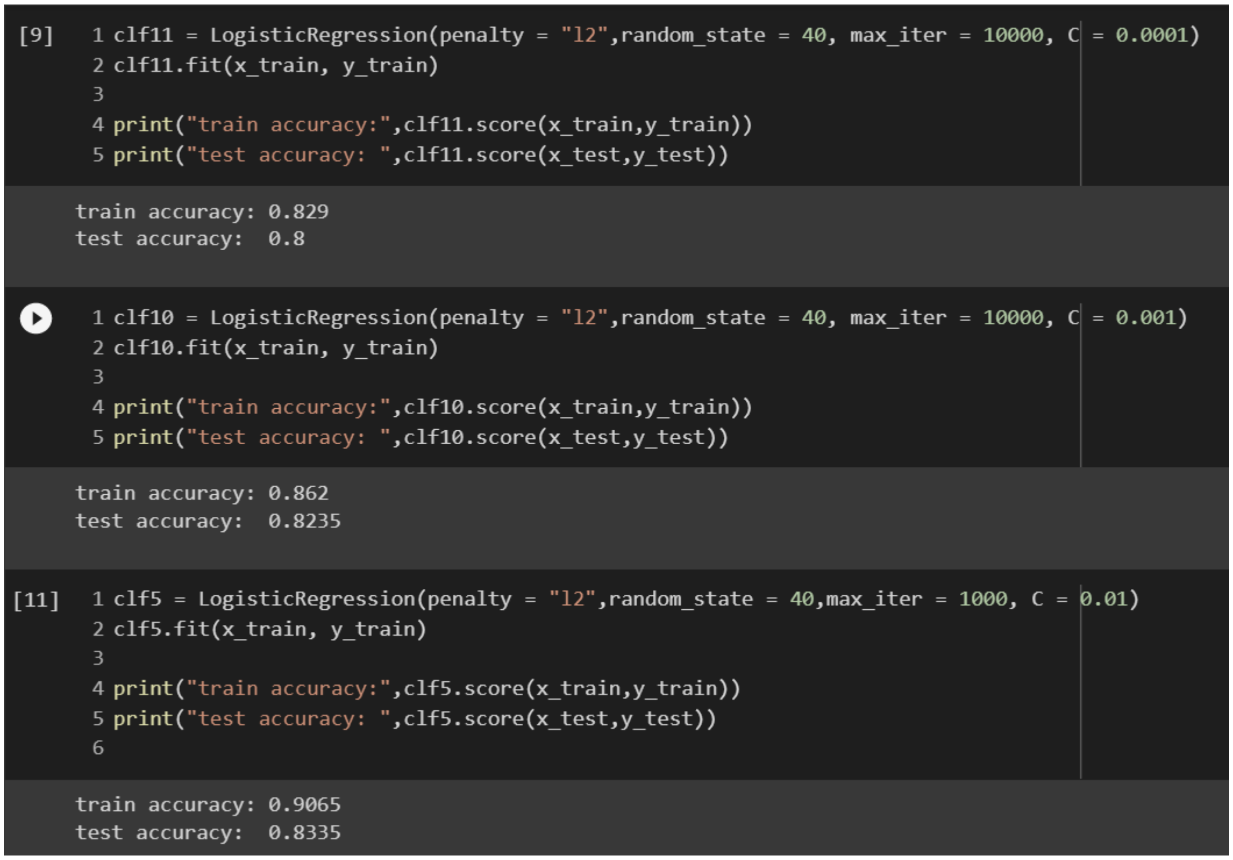
```python
clf13 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 10000, C = 0.000001)
clf13.fit(x_train, y_train)

print("train accuracy:",clf13.score(x_train,y_train))
print("test accuracy: ",clf13.score(x_test,y_test))
```

```
train accuracy: 0.8035
test accuracy:  0.781
```

[8]

```python
clf12 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 10000, C = 0.00001)
clf12.fit(x_train, y_train)

print("train accuracy:",clf12.score(x_train,y_train))
print("test accuracy: ",clf12.score(x_test,y_test))
```

```
train accuracy: 0.818
test accuracy:  0.7895
```



[9]

```python
clf11 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 10000, C = 0.0001)
clf11.fit(x_train, y_train)

print("train accuracy:",clf11.score(x_train,y_train))
print("test accuracy: ",clf11.score(x_test,y_test))
```

```
train accuracy: 0.829
test accuracy:  0.8
```

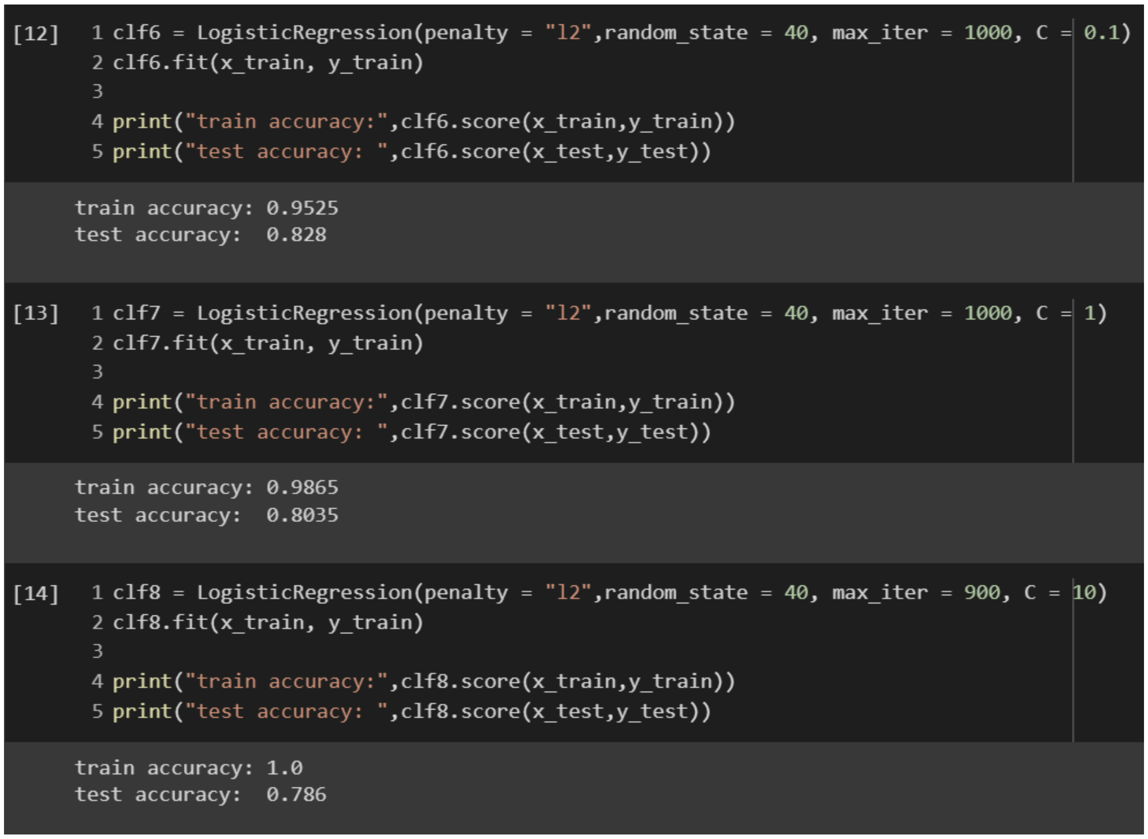
```python
clf10 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 10000, C = 0.001)
clf10.fit(x_train, y_train)

print("train accuracy:",clf10.score(x_train,y_train))
print("test accuracy: ",clf10.score(x_test,y_test))
```

```
train accuracy: 0.862
test accuracy:  0.8235
```

[11]

```python
clf5 = LogisticRegression(penalty = "l2",random_state = 40,max_iter = 1000, C = 0.01)
clf5.fit(x_train, y_train)

print("train accuracy:",clf5.score(x_train,y_train))
print("test accuracy: ",clf5.score(x_test,y_test))

```

```
train accuracy: 0.9065
test accuracy:  0.8335
```



```
[12] clf6 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 1000, C = 0.1)
     clf6.fit(x_train, y_train)

     print("train accuracy:",clf6.score(x_train,y_train))
     print("test accuracy: ",clf6.score(x_test,y_test))
```

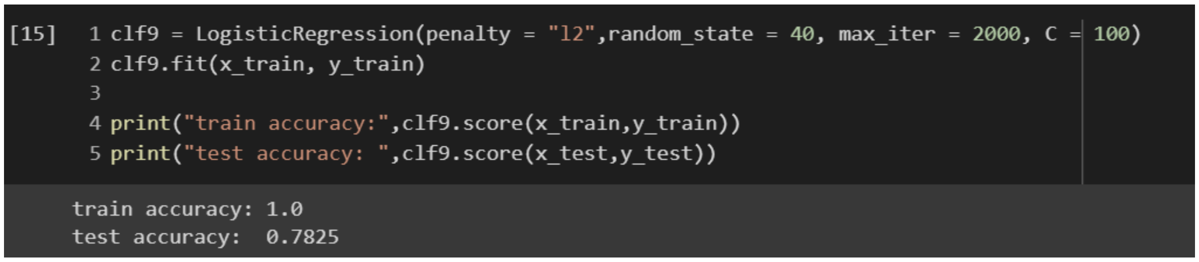
```
train accuracy: 0.9525
test accuracy:  0.828
```

```
[13] clf7 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 1000, C = 1)
     clf7.fit(x_train, y_train)

     print("train accuracy:",clf7.score(x_train,y_train))
     print("test accuracy: ",clf7.score(x_test,y_test))
```

```
train accuracy: 0.9865
test accuracy:  0.8035
```

```
[14] clf8 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 900, C = 10)
     clf8.fit(x_train, y_train)

     print("train accuracy:",clf8.score(x_train,y_train))
     print("test accuracy: ",clf8.score(x_test,y_test))
```

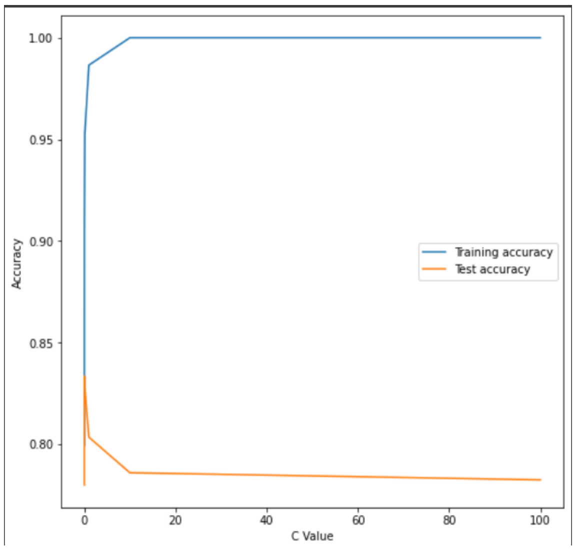
```
train accuracy: 1.0
test accuracy:  0.786
```



```
[15] clf9 = LogisticRegression(penalty = "l2",random_state = 40, max_iter = 2000, C = 100)
     clf9.fit(x_train, y_train)

     print("train accuracy:",clf9.score(x_train,y_train))
     print("test accuracy: ",clf9.score(x_test,y_test))
```

```
train accuracy: 1.0
test accuracy:  0.7825
```

Plot of training and test accuracy calculated above:

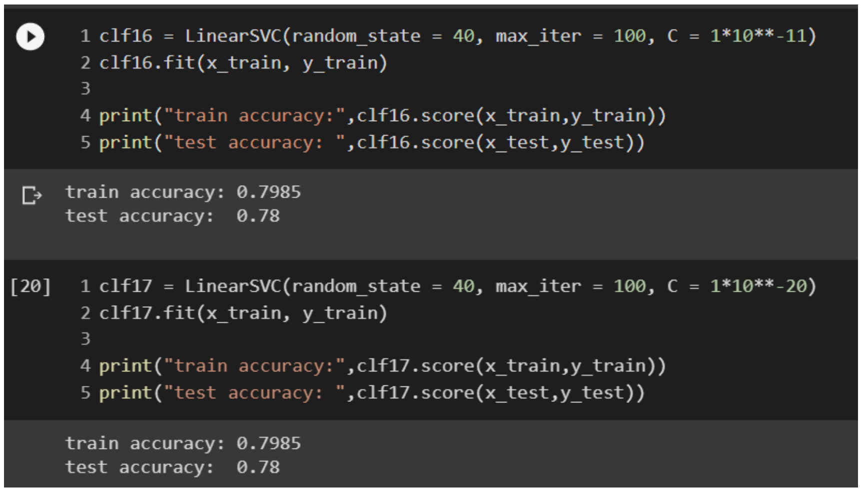


It's hard to tell from the graph but it's obvious from my calculations that as my regularization parameter increases the overfitting increases even by a small amount. And there's no underfitting from the calculations I did. This means regardless of C the training accuracy will never be higher than the training set accuracy. This makes it impossible for us find that sweet pot where the two (training acc and test acc) meet because they never do. So for this model the best we could do is choose a regularization parameter that maximizes the values of the training and test accuracy and minimizes the overfitting of the two.

When the regularization parameter is very low there is little to no overfitting. For example when C = 0.000000000000000000001 both training and test accuracy are almost the same however, they have a low accuracy level of 0.5.

Therefore, I personally think that the best value for C is 0.0001 as the result achieved from this was the highest possible training and test accuracy (0.829 & 0.8 respectively) one could get for this model while keeping the overfitting to a minimum.

Second : Linear SVM



```
clf16 = LinearSVC(random_state = 40, max_iter = 100, C = 1*10**-11)
clf16.fit(x_train, y_train)

print("train accuracy:",clf16.score(x_train,y_train))
print("test accuracy: ",clf16.score(x_test,y_test))
```

```
train accuracy: 0.7985
test accuracy:  0.78
```

```
[20]
clf17 = LinearSVC(random_state = 40, max_iter = 100, C = 1*10**-20)
clf17.fit(x_train, y_train)

print("train accuracy:",clf17.score(x_train,y_train))
print("test accuracy: ",clf17.score(x_test,y_test))
```

```
train accuracy: 0.7985
test accuracy:  0.78
```

Similar to linear regression after some point no matter how low C is the training and test accuracy becomes constant. But not as low as what it would have been in linear regression.

Below each plot represents a different range of C. This is because I want to explore the range of C that gives the highest training and test accuracy with a low level of overfitting

C_vals = np.arange(0.0009,0.001,0.00001)
Test Accuracy : 0.83 (approx)
Training accuracy : 0.905 (approx)
Overfitting = 0.075

C_vals = np.arange(0.00009,0.0001,0.000001)
Test Accuracy : 0.825 (approx)
Training accuracy : 0.860 (approx)
Overfitting = 0.035

C_vals = np.arange(0.000009,0.00001,0.0000001)
Test Accuracy : 0.8 (approx)
Training accuracy : 0.828 (approx)
Overfitting = 0.028

C_vals = np.arange(0.0000009,0.000001,0.00000001)
Test Accuracy : 0.786 (approx)
Training accuracy : 0.818 (approx)
Overfitting = 0.032

C_vals = np.arange(0.00000009,0.0000001,0.000000001)
Towards the end of this range at around C = 0.000000098:
Test Accuracy : 0.7815 (approx)
Training accuracy : 0.8035 (approx)
Overfitting = 0.022

Roughly as the range of C values increase the overfitting increase. The best range for the c value we can take in my opinion is a value between 0.000009 and 0.00001. So C = 0.0000095 would be the best possible the regularization paameter can have for this dataset and chosen model.

Third : K-fold

The code provided in the google colab file performs k-fold cross-validation to tune the hyperparameters (C) for two classification models, Linear SVM and Logistic Regression. It uses the training data to divide it into k-folds and trains each model on k-1 folds while validating on the remaining fold. This process is repeated for different values of C, and the average validation score for each C value is computed. The code then selects the value of C that gives the highest validation score for each model and returns them. Finally, the optimal models are trained on the entire training set and tested on a separate test set to report their test error.

When K was given the value of 10

Optimal regularization parameter for Logistic Regresion	: 0.01
Optimal regularization parameter for SVM	: 0.0001

Trained each model using the entire training set and evaluated their performance on the test set.
SVM test error: 0.15400000000000003
Logistic regression test error: 0.15400000000000003

Determined if the test errors are significantly different by calculating the confidence intervals.
SVM 95% confidence interval: (0.022441963309851237, 0.2855580366901488)
Logistic regression 95% confidence interval: (0.022441963309851237, 0.2855580366901488)

The SVM and logistic regression models have similar test error rates, with both having a test error of approximately 0.15. The confidence intervals for the test errors of both models overlap, indicating that the difference in mean test errors between the models is not statistically significant at a 95% confidence level.

Fourth : SVM with Gaussian Kernel

Gamma: 1e-09
Optimal C: 1e-08
SVM test error: 0.21750000000000003
SVM training error: 0.2004166666666667

Gamma: 1e-08
Optimal C: 1e-08
SVM test error: 0.21750000000000003
SVM training error: 0.20025000000000004

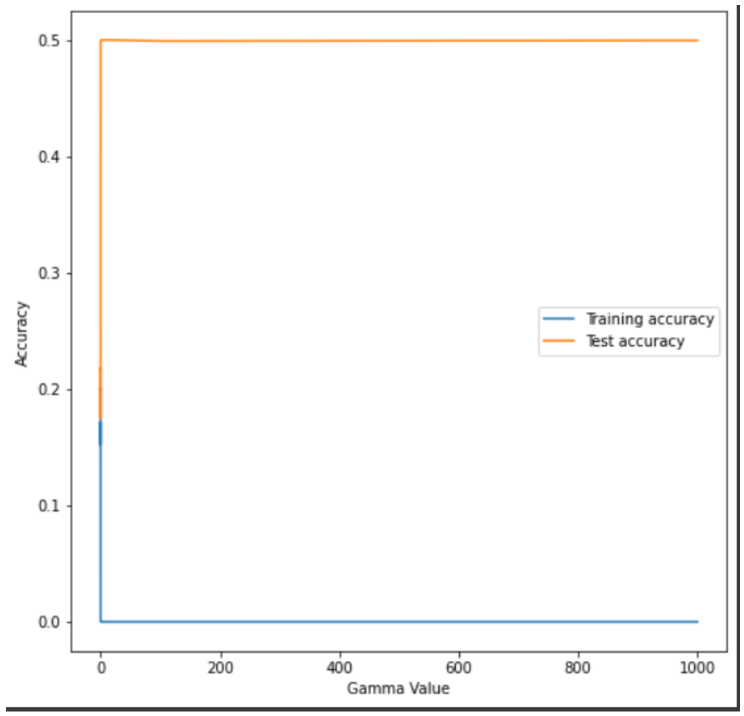
Gamma: 1e-07
Optimal C: 1e-08
SVM test error: 0.21799999999999997
SVM training error: 0.2004166666666667

Gamma: 1e-06
Optimal C: 1e-08
SVM test error: 0.21799999999999997
SVM training error: 0.20016666666666671

```
Gamma:  1e-05
Optimal C:  1e-08
SVM test error: 0.21350000000000002
SVM training error: 0.20066666666666666


Gamma:  0.0001
Optimal C:  0.1
SVM test error: 0.1995
SVM training error: 0.1795


Gamma:  0.001
Optimal C:  0.1
SVM test error: 0.1745
SVM training error: 0.15158333333333333


Gamma:  0.01
Optimal C:  0.1
SVM test error: 0.21950000000000003
SVM training error: 0.17200000000000004


Gamma:  0.1
Optimal C:  1e-08
SVM test error: 0.49850000000000005
SVM training error: 0.0


Gamma:  1.0
Optimal C:  1e-08
SVM test error: 0.5
SVM training error: 0.0


Gamma:  10.0
Optimal C:  1e-08
SVM test error: 0.5
SVM training error: 0.0
```

```
Gamma:  100.0
Optimal C:  1e-08
SVM test error: 0.499
SVM training error: 0.0
```

```
Gamma:  1000.0
Optimal C:  1e-08
SVM test error: 0.49950000000000006
SVM training error: 0.0
```



Again it's not obvious from the plot but from from the calculations above we can conclude that as the gamma value increase the test error increase to as high as 0.5 and training error reduces to as low as 0. The value for gamma that give the lowest amount of overfitting while keeping the errors at aminimum in my opinion is Gamma: 1e-05.

The test error for the linear svm model was much lower than for the kernalised SVM model.