Font Character Multi-Output Classification Problem Using SVM:

This is a multi-class classification problem. But SVM doesn't support multiclass classification. For multiclass classification, we need to break down the multi-classification problem into smaller subproblems, all of which are binary classification problems.

The popular methods which are used to perform multi-classification on the problem statements using SVM are as follows: One vs One (OVO) approach

And One vs All (OVA) approach. Here, I used the one vs all method.

This problem is a multi-input and multi-output classification issue that we would like to solve by the SVM algorithm. This is a classification problem that involves 26 classes as outputs and 6 features or inputs. So, this is a 6-dimensional problem. The task here is to design and train an SVM to minimize the error between actual and predicted output values. We have 78 data points as the training dataset and 78 data points as the testing dataset. The steps I did from beginning to end will be explained in the following. Note: Here, data is not linearly separable, so, we need to combine SVM with kernels that help SVM become extremely powerful. Although we have 26 outputs or classes, after encoding these outputs, we have a multi-class classification problem that needs to solve as 26 binary classification problems.

1. **Import required libraries and Load data from local drive and load dataset**

```
[ ] # Standard imports
    import numpy as np
    import matplotlib.pyplot as plt
    from scipy import stats
    import pandas as pd
    %matplotlib inline

    # Use seaborn plotting defaults
    import seaborn as sns; sns.set()
```

```
#Load data from local drive
from google.colab import files
uploaded = files.upload()
```

```
Choose Files 2 files
• font_test.csv(application/vnd.ms-excel) - 6486 bytes, last modified: 3/13/2019 - 100% done
• font_train.csv(application/vnd.ms-excel) - 6486 bytes, last modified: 3/13/2019 - 100% done
Saving font_test.csv to font_test.csv
Saving font_train.csv to font_train.csv
```

```
[ ] #Load data set
    font_train = pd.read_csv('font_train.csv')
    font_test = pd.read_csv('font_test.csv')
```

2. **See how the train data is:**

```
font_train.head()
```

	input1	input2	input3	input4	input5	input6	input7	input8	input9	input10	...	Q	R	S	T	U	V	W	X	Y	Z
0	8	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	12	10	1	1	0	0	0	4	6	0	...	0	0	0	0	0	0	0	0	0	0
3	21	10	4	4	0	1	1	0	5	0	...	0	0	0	0	0	0	0	0	0	0
4	27	12	3	3	0	8	0	5	0	2	...	0	0	0	0	0	0	0	0	0	0

5 rows × 40 columns

3. **Exploratory data analysis**

Now, I will explore the data to gain insights about the data.

```
# view dimensions of dataset
font_train.shape
```

```
(78, 40)
```

```
font_test.shape
```

```
(78, 40)
```

```
font_train.describe()
```

	input_a	input_b	input_c	input_d	input_e	input_f	input_g	input_h	input_i	input_j	...	Q	R	S	T	
count	78.000000	78.000000	78.000000	78.000000	78.000000	78.000000	78.000000	78.000000	78.000000	78.000000	...	78.000000	78.000000	78.000000	78.000000	78.
mean	8.089744	11.358974	2.307692	2.525641	0.192308	1.884615	0.358974	1.307692	1.166667	0.397436	...	0.038462	0.038462	0.038462	0.038462	0.
std	7.552734	6.907505	2.549804	2.930912	0.967762	3.057801	1.459179	2.034174	2.635505	0.671106	...	0.193552	0.193552	0.193552	0.193552	0.
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.
25%	0.000000	9.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.
50%	8.000000	11.000000	2.000000	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.
75%	12.750000	14.750000	4.000000	4.000000	0.000000	4.000000	0.000000	3.500000	0.000000	1.000000	...	0.000000	0.000000	0.000000	0.000000	0.
max	27.000000	31.000000	10.000000	12.000000	5.000000	11.000000	8.000000	5.000000	9.000000	3.000000	...	1.000000	1.000000	1.000000	1.000000	1.

8 rows × 40 columns

4. Our target variables are 'A','B',...,'Z' columns. So, I will check its distribution. We will check target distribution to know whether we have class imbalanced or not.

```
# check distribution of target_class column
#We have three different alphabets for training data, so, we have three 'A', three 'B',...,three 'Z'

font_train['A'].value_counts()
font_train['Z'].value_counts()
```

```
0    75
1     3
Name: Z, dtype: int64
```

```
# view the percentage distribution of target_class column

font_train['A'].value_counts()/float(len(font_train))
```

```
0    0.961538
1    0.038462
Name: A, dtype: float64
```

We can see that percentage of observations of the class label 0 and 1 for each label is 96.15% and 3.84%. This distribution is applied for all the other target variables, so, I didn't repeat showing distribution results of the others. As a result, this is a class imbalanced problem.

5. view summary of dataset

```
# view summary of dataset

font_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 78 entries, 0 to 77
Data columns (total 40 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   input_a  78 non-null     int64
 1   input_b  78 non-null     int64
 2   input_c  78 non-null     int64
 3   input_d  78 non-null     int64
 4   input_e  78 non-null     int64
 5   input_f  78 non-null     int64
 6   input_g  78 non-null     int64
 7   input_h  78 non-null     int64
 8   input_i  78 non-null     int64
 9   input_j  78 non-null     int64
 10  input_k  78 non-null     int64
 11  input_l  78 non-null     int64
 12  input_m  78 non-null     int64
 13  input_n  78 non-null     int64
 14  A        78 non-null     int64
 15  B        78 non-null     int64
 16  C        78 non-null     int64
 17  D        78 non-null     int64
 18  E        78 non-null     int64
 19  F        78 non-null     int64
 20  G        78 non-null     int64
 21  H        78 non-null     int64
 22  I        78 non-null     int64
 23  J        78 non-null     int64
 24  K        78 non-null     int64
 25  L        78 non-null     int64
```

```
# view summary of dataset

font_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 78 entries, 0 to 77
Data columns (total 40 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   input_a  78 non-null     int64
 1   input_b  78 non-null     int64
 2   input_c  78 non-null     int64
 3   input_d  78 non-null     int64
 4   input_e  78 non-null     int64
 5   input_f  78 non-null     int64
 6   input_g  78 non-null     int64
 7   input_h  78 non-null     int64
 8   input_i  78 non-null     int64
 9   input_j  78 non-null     int64
 10  input_k  78 non-null     int64
 11  input_l  78 non-null     int64
 12  input_m  78 non-null     int64
 13  input_n  78 non-null     int64
 14  A        78 non-null     int64
 15  B        78 non-null     int64
 16  C        78 non-null     int64
 17  D        78 non-null     int64
 18  E        78 non-null     int64
 19  F        78 non-null     int64
 20  G        78 non-null     int64
 21  H        78 non-null     int64
 22  I        78 non-null     int64
 23  J        78 non-null     int64
```

We can see that there are no missing values in the dataset and all the variables are numerical variables.

Summary of numerical variables

There are 40 numerical variables in each train and test datasets.

14 are continuous variables and 26 are discrete variable.

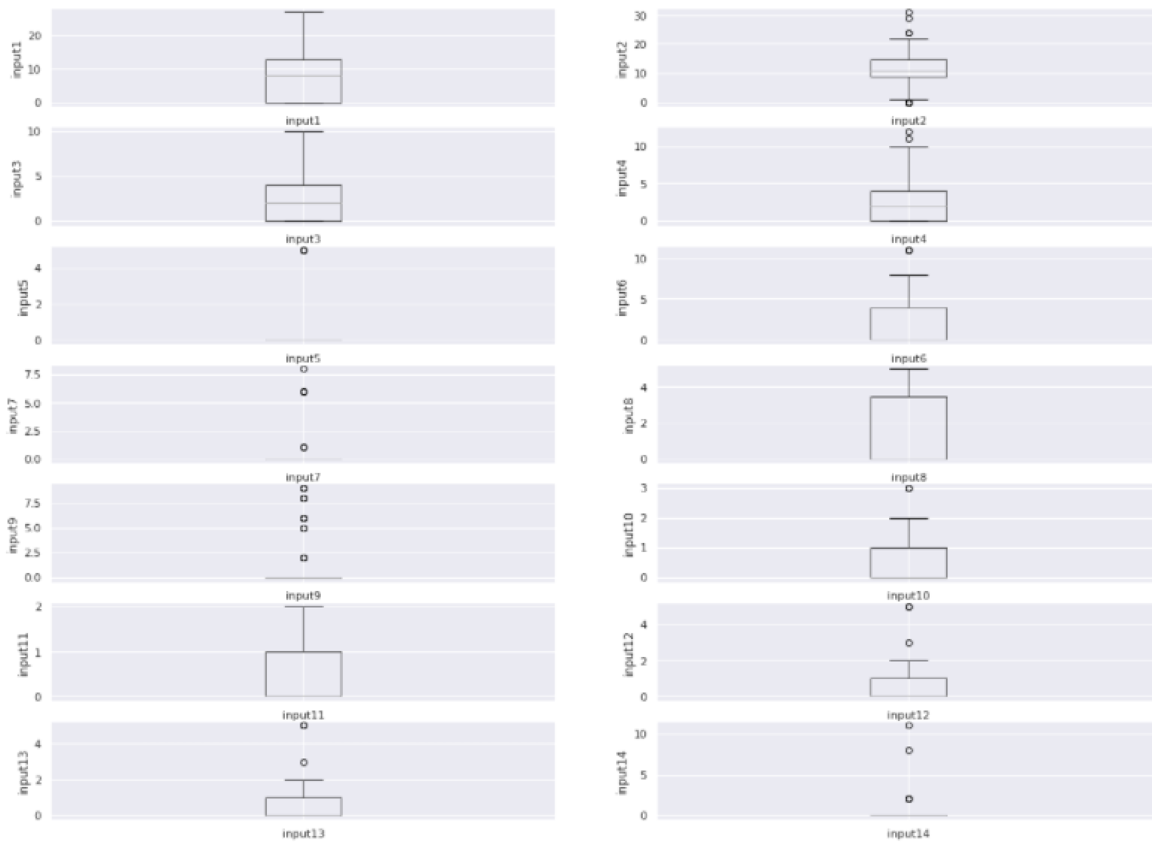
The discrete variables are target variables.

There are no missing values in the dataset.

6. **Outliers:**
   On closer inspection, we can suspect that all the continuous variables may contain outliers. I will draw boxplots to visualize outliers in the above variables.

```
# draw boxplots to visualize outliers

plt.figure(figsize=(20,15))


plt.subplot(7, 2, 1)
fig = font_train.boxplot(column='input1')
fig.set_title('')
fig.set_ylabel('input1')


plt.subplot(7, 2, 2)
fig = font_train.boxplot(column='input2')
fig.set_title('')
fig.set_ylabel('input2')


plt.subplot(7, 2, 3)
fig = font_train.boxplot(column='input3')
fig.set_title('')
fig.set_ylabel('input3')


plt.subplot(7, 2, 4)
fig = font_train.boxplot(column='input4')
fig.set_title('')
fig.set_ylabel('input4')

plt.subplot(7, 2, 5)
```

```
Text(0, 0.5, 'input14')
```



The above boxplots confirm that there are a lot of outliers in these variables.

Handle outliers with SVM:

There are 2 variants of SVMs. They are hard-margin variant of SVM and soft-margin variant of SVM.

One version of SVM is called soft-margin variant of SVM. In this case, we can have a few points incorrectly classified or classified with a margin less than 1. But for every such point, we have to pay a penalty in the form of C parameter, which controls the outliers. Here, the message is that since the dataset contains outliers, so the value of C should be high while training the model and the margin should be soft.

7. **Check the distribution of variables**

   Now, I will plot the histograms to check distributions to find out if they are normal or skewed.

```
sns.pairplot(font_train[["input1",'input2','input3','input4','input5','input6','input7','input8','input9','input10','input11','input12','input13','input14']])
```

```
<seaborn.axisgrid.PairGrid at 0x7fc81aecae90>
```

We can see that all the 14 continuous variables are skewed. So, we need to perform normalization.

8. Declare feature vector and target variable

```
# Separate feature and target variables for train dataset
font_train = pd.read_csv('font_train.csv')
font_test = pd.read_csv('font_test.csv')
X_train = font_train.copy()
y_train = X_train.iloc[:, 14:40]
print(y_train.shape)
```

```
(78, 26)
```

```
# Separate feature and target variables for train dataset
X_train.drop(['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'], axis = 1, inplace = True)
print(X_train.shape)
```

```
(78, 14)
```

```
# Separate feature and target variables for test dataset
font_train = pd.read_csv('font_train.csv')
font_test = pd.read_csv('font_test.csv')
X_test = font_test.copy() # X_test is a dataframe
y_test = X_test.iloc[:, 14:40]
print(y_test.shape)
```

```
(78, 26)
```

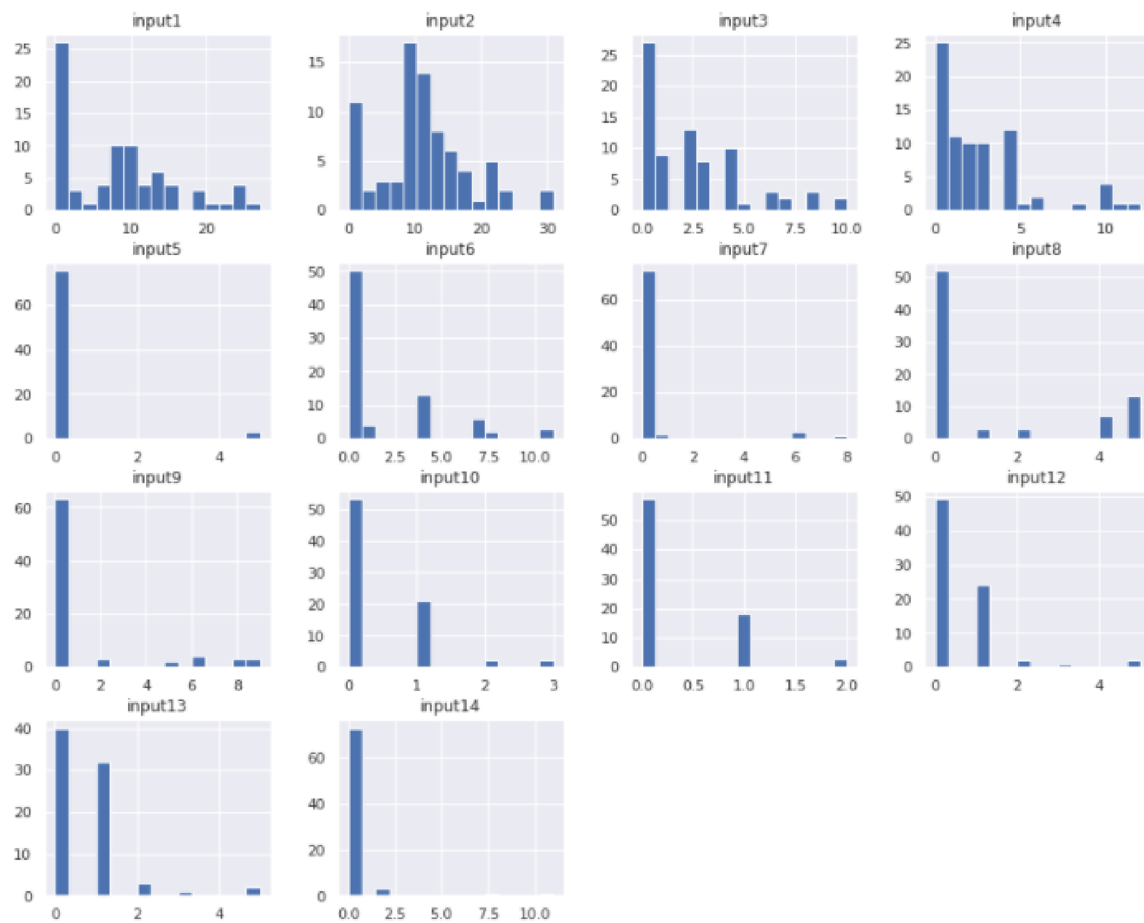
```
# Separate feature and target variables for test dataset
X_test.drop(['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'], axis = 1, inplace = True)
print(X_test.shape)
```

```
(78, 14)
```

9. **Check the distribution of variables**

   Now, I will plot the histograms to check distributions to find out if they are normal or skewed.

```python
# plot histogram to check distribution

X_train.hist(figsize=(15,12),bins = 15)
plt.title("Features Distribution")
plt.show()
```



Also, here, We can see that all the 14 continuous variables are skewed.

10. **Feature Scaling**

A standardization is scaling features to lie between a given minimum and maximum value, often between zero and one, or so that the maximum absolute value of each feature is scaled to unit size. This can be achieved using MinMaxScaler or MaxAbsScaler, respectively. The motivation to use this scaling include robustness to very small standard deviations of features and preserving zero entries in sparse data.

```
[ ] cols = X_train.columns
```

```
[ ] from sklearn import preprocessing
    from sklearn.preprocessing import MinMaxScaler
```

```
[ ] scaler = preprocessing.MinMaxScaler()
    X_train = scaler.fit_transform(X_train) #fit means training the model which is done on training data, by using this line X_train will be an array not a dataframe
    X_train.dtype

    dtype('float64')
```

The same instance of the transformer can then be applied to some new test data unseen during the fit call: the same scaling and shifting operations will be applied to be consistent with the transformation performed on the train data:

```
X_test = scaler.transform(X_test) #The output is an array
X_test
X_test.dtype

dtype('float64')
```

```
[ ] X_train = pd.DataFrame(X_train, columns=[cols]) #Convert numpy array to dataframe, we do this since we want to add this dataframe to the other dataframe and make a new dataframe
```

11. Run SVM with default hyperparameters

We have support vector classifiers. I will use two of them: 'linear' and 'RBF' classifiers

```
# import SVC classifier
from sklearn.svm import SVC


# import metrics to compute accuracy
from sklearn.metrics import accuracy_score

for i in range(0,25):
# instantiate classifier with default hyperparameters
  svc=SVC()


# fit classifier to training set
  svc.fit(X_train,y_train.iloc[:,i])


# make predictions on test set
  y_pred=svc.predict(X_test)


# compute and print accuracy score
  print('Model accuracy score with default hyperparameters: {0:0.4f}'. format(accuracy_score(y_test.iloc[:,i], y_pred)))
```

```
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
```

Multi-class classification problem was solved as multiple binary classification problems by using for loop(instead we could use one vs rest or one vs one methods). Model accuracy score with default hyperparameters for different target columns are: 0.9615, 0.9872, 1

12. **Run SVM with RBF kernel and C=100.0**

    We have seen that there are outliers in our dataset. So, we should increase the value of C as higher C means fewer outliers. So, I will run SVM with kernel=rbf and C=100.0.

```python
# import SVC classifier
from sklearn.svm import SVC


# import metrics to compute accuracy
from sklearn.metrics import accuracy_score

for i in range(0,25):
# instantiate classifier with default hyperparameters
  svc=SVC(C=100.0)


# fit classifier to training set
  svc.fit(X_train,y_train.iloc[:,i])


# make predictions on test set
  y_pred=svc.predict(X_test)
  y_pred_test=svc.predict(X_test)

# compute and print accuracy score
  print('Model accuracy score with default hyperparameters: {0:0.4f}'. format(accuracy_score(y_test.iloc[:,i], y_pred)))
```

Model accuracy score with RBF kernel and C=100.0 : 0.9744, 0.9832,1

We can see that we obtain higher accuracy with C=100.0 as higher C means fewer outliers and accuracy is made better for almost all the classes except one of them.

Now, I will further increase the value of C=1000.0 and check accuracy.

I also do linear kernel with C=1, 100, and 1000

13. **Run SVM with linear kernel and C=1.0:**

```python
# import SVC classifier
from sklearn.svm import SVC


# import metrics to compute accuracy
from sklearn.metrics import accuracy_score

for i in range(0,25):
# instantiate classifier with default hyperparameters
  svc=SVC(kernel='linear', C=1.0)


# fit classifier to training set
  svc.fit(X_train,y_train.iloc[:,i])


# make predictions on test set
  y_pred=svc.predict(X_test)


# compute and print accuracy score
  print('Model accuracy score with default hyperparameters: {0:0.4f}'. format(accuracy_score(y_test.iloc[:,i], y_pred)))
```

```
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9487
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
Model accuracy score with default hyperparameters: 0.9615
```

14. **Run SVM with linear kernel and C=100.0**

```python
# import SVC classifier
from sklearn.svm import SVC


# import metrics to compute accuracy
from sklearn.metrics import accuracy_score

for i in range(0,25):
# instantiate classifier with default hyperparameters
  linear_svc=SVC(kernel='linear', C=100.0)


# fit classifier to training set
  linear_svc.fit(X_train,y_train.iloc[:,i])


# make predictions on test set
  y_pred=linear_svc.predict(X_test)


# compute and print accuracy score
  print('Model accuracy score with default hyperparameters: {0:0.4f}'. format(accuracy_score(y_test.iloc[:,i], y_pred)))
```

```
Model accuracy score with default hyperparameters: 0.9744
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9744
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9487
Model accuracy score with default hyperparameters: 0.9744
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 1.0000
Model accuracy score with default hyperparameters: 0.9872
Model accuracy score with default hyperparameters: 0.9359
Model accuracy score with default hyperparameters: 0.9744
```

We see that we can obtain higher accuracy with C=100.0 and C=1000.0 as compared to C=1.0. But 'RBF' kernel has better performance than the linear for this non-linear separable dataset.

15. **Compare the train-set and test-set accuracy**

Now, I will compare the train-set and test-set accuracy to check for overfitting.

```python
y_pred_train = linear_svc.predict(X_train)

y_pred_train
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:1692: FutureWarning: Feature names only support names that are all strings. Got feature names with dtypes: ['tuple']. An error will be rai
  FutureWarning,
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0])
```

```python
for i in range(0,25):
  print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train.iloc[:,i], y_pred_train)))
```

```
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9487
Training-set accuracy score: 0.9744
```

We can see that the training set accuracies for all classes are lower than test-set accuracies. So, there is no overfitting.

To show it better, I will check for overfitting and underfitting by comparing the train-set and test-set accuracy to check for overfitting.

```python
# print the scores on training and test set
for i in range(0,25):
  print('Training set score: {:.4f}'.format(svc.score(X_train, y_train.iloc[:,i])))

  print('Test set score: {:.4f}'.format(svc.score(X_test, y_test.iloc[:,i])))
```

```
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
Training set score: 0.9615
Test set score: 0.9615
```

16. Compare model accuracy with null accuracy

So, the model accuracy is 0.9231. But, we cannot say that our model is very good based on the above accuracy. We must compare it with the null accuracy. Null accuracy is the accuracy that could be achieved by always predicting the most frequent class. So, we should first check the class distribution in the test set.

```python
# check class distribution in test set
for i in range(0,25):
  print(y_test.iloc[:,i].value_counts())
```

```
0    75
1     3
Name: A, dtype: int64
0    75
1     3
Name: B, dtype: int64
0    75
1     3
Name: C, dtype: int64
0    75
1     3
Name: D, dtype: int64
0    75
1     3
Name: E, dtype: int64
0    75
1     3
Name: F, dtype: int64
0    75
1     3
Name: G, dtype: int64
0    75
1     3
Name: H, dtype: int64
0    75
1     3
Name: I, dtype: int64
0    75
1     3
Name: J, dtype: int64
0    75
1     3
Name: K, dtype: int64
0    75
1     3
Name: L, dtype: int64
0    75
1     3
Name: M, dtype: int64
0    75
1     3
Name: N, dtype: int64
```

We can see that the occurrences of most frequent class 0 are 75 for each class. So, we can calculate null accuracy by dividing 75 by a total number of occurrences.

```
[ ]  # check null accuracy score

     null_accuracy = (75/(75+3))

     print('Null accuracy score: {0:0.4f}'. format(null_accuracy))

     Null accuracy score: 0.9615
```

We can see that our model accuracy score is 0.9231 but the null accuracy score is 0.9615. So, we can conclude that our SVM classifier is doing a very bad job in predicting the class labels.

17. Comments

We get maximum accuracy with rbf kernel with C=100.0. and the accuracy is 0.9231. Based on the above analysis we can conclude that our classification model accuracy is very bad. Our model is doing a very bad job in terms of predicting the class labels.

But, we should note that here, we have an imbalanced dataset. The problem is that accuracy is an inadequate measure for quantifying predictive performance in the imbalanced dataset problem.

So, we must explore alternative metrices that provide better guidance in selecting models. In particular, we would like to know the underlying distribution of values and the type of errors our classifer is making. One such metric to analyze the model performance in imbalanced classes problem is Confusion matrix.

```
# Print the Confusion Matrix and slice it into four pieces

from sklearn.metrics import confusion_matrix
for i in range(0,25):
  y_test1 = y_test.iloc[:,i]
  cm = confusion_matrix(y_test1, y_pred_test)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

```
Confusion matrix

 [[74  1]
 [ 1  2]]

True Positives(TP) =  74

True Negatives(TN) =  2

False Positives(FP) =  1

False Negatives(FN) =  1
```

The confusion matrix shows 74 + 2 = 76 correct predictions and 1 + 1 = 2 incorrect predictions.

In this case, we have:

True Positives (Actual Positive:1 and Predict Positive:1) - 74

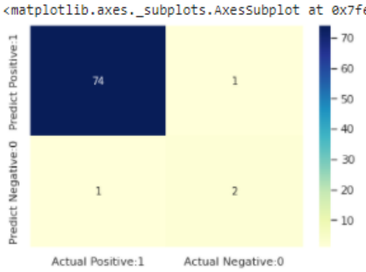
True Negatives (Actual Negative:0 and Predict Negative:0) - 2

False Positives (Actual Negative:0 but Predict Positive:1) - 1 (Type I error)

False Negatives (Actual Positive:1 but Predict Negative:0) - 1 (Type II error)

```
# visualize confusion matrix with seaborn heatmap

cm_matrix = pd.DataFrame(data=cm, columns=['Actual Positive:1', 'Actual Negative:0'],
                                 index=['Predict Positive:1', 'Predict Negative:0'])

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fefc1a816d0>
```



18. Support Vectors:

In Scikit-Learn, the identity of these points are stored in the support_vectors_ attribute of the classifier:

```
[34] svc.support_vectors_

array([[0.        , 0.32258065, 0.1       , 0.08333333, 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.2       , 0.        ],
       [0.        , 0.64516129, 0.4       , 0.33333333, 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        ],
       [0.11111111, 0.51612903, 0.3       , 0.25      , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.2       , 0.2       , 0.        ],
       [0.        , 0.12903226, 0.4       , 0.33333333, 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.4       , 0.4       , 0.        ],
       [0.        , 0.32258065, 0.3       , 0.25      , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.2       , 0.2       , 0.        ]])
```

Number of Support Vectors:

```
print(svc.support_vectors_.shape)

(5, 14)
```

So, the number of support vectors that rbf classifier was found is 5.