Support Vector Machines: Non Separable classification - Multi-Font Character Recognition

Author: Rajpal Virk | 12 March 2020

Introduction

A Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems and also known as discriminative classifier. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes well.

Dataset

The data consists of 6 fonts (Courier, New York, Chicago, Geneva, Times, and Venice), was collected and quantized by Lee [1988]. A brief summary of the data collection method is presented here [Logar et al., 1993]:

1. The image of the letter is normalized to an 18 x 18 character matrix, where the line thickness is one and the image is represented by 0's (background) and 1's (foreground).
2. Fourteen properties similar to those proposed by Fujii and Morita (see reference) were extracted from each image. Each property is a 3 x 3 matrix, thus, for each image, a 14 x 9 matrix is generated. This is the X matrix for that image. A property recognition matrix, Y, is constructed for each image and is also a 14 x 9 matrix. It is chosen arbitrarily and is as simple as possible. W is a 9 x 9 filter matrix which maps X to Y and can be found from: W = X Y, where X the pseudo-inverse of X.
3. A 3 x 3 window is moved from upper left to lower right over the character image. The 9 elements in the window are multiplied by the matrix W. If the output matches a row of Y, say row k, the kth place in the count matrix is incremented by the weighting factor of that property.

Thus, the count matrix for a character contains the number of exact template matches, weighted by position. The result is 156 (26 x 6) 14-element vectors.

Train Data

Train data contains Courier, New York, and Chicago Fonts; 78 Patterns in Total (26 x 3).

- Length of the Input Per Pattern: 14
- Length of the Output Per Pattern: 26 (Binary Values Check Alphabet Match: 0 denotes "no match" and 1 denotes "match")

Sequence:

- Pattern #1: Courier Font Alphabet A
- Pattern #2: New York Font Alphabet A
- Pattern #3: Chicago Font Alphabet A
- Pattern #4: Courier Font Alphabet B
- Pattern #5: New York Font Alphabet B
- Pattern #6: Chicago Font Alphabet B....
- Pattern #76: Courier Font Alphabet Z
- Pattern #77: New York Font Alphabet Z
- Pattern #78: Chicago Font Alphabet Z

Test Data

Test data contains Geneva, Times, and Venice Fonts; 78 Patterns in Total (26 x 3).

- Length of the Input: 14
- Length of the Output Per Pattern: 26 (Binary Values Check Alphabet Match: 0 denotes "no match" and 1 denotes "match") Length of the Pattern: 40

Sequence:

- Pattern #1: Geneva Font Alphabet A
- Pattern #2: Times Font Alphabet A
- Pattern #3: Venice Font Alphabet A
- Pattern #4: Geneva Font Alphabet B
- Pattern #5: Times Font Alphabet B
- Pattern #6: Venice Font Alphabet B ...
- Pattern #76: Geneva Font Alphabet Z
- Pattern #77: Times Font Alphabet Z
- Pattern #78: Venice Font Alphabet Z

Methodology

Methodology that we'll follow is as below:

1. Load, Review and Pre-process data
2. Build Model
3. Train Model
4. Evaluate Model
5. Conclusion

Since, we have sparse dataset, we'll use stratified shuffle split method for cross-validation to split data.

Load, Review and Pre-process data

In [1]:
```
# Import required libraries

from IPython.core.display import display
import pandas as pd
import numpy as np

from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import StratifiedShuffleSplit

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# Load data
df_train = pd.read_excel('data.xlsx' ,sheet_name = 'train', header=None)
df_test = pd.read_excel('data.xlsx' ,sheet_name = 'test', header=None)
print('Train data:')
display(df_train.head())
print()
print('Test data:')
display(df_train.head())
```

Train data:

	0	1	2	3	4	5	6	7	8	9	...	30	31	32	33	34	35	36	37	38	39
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

Test data:

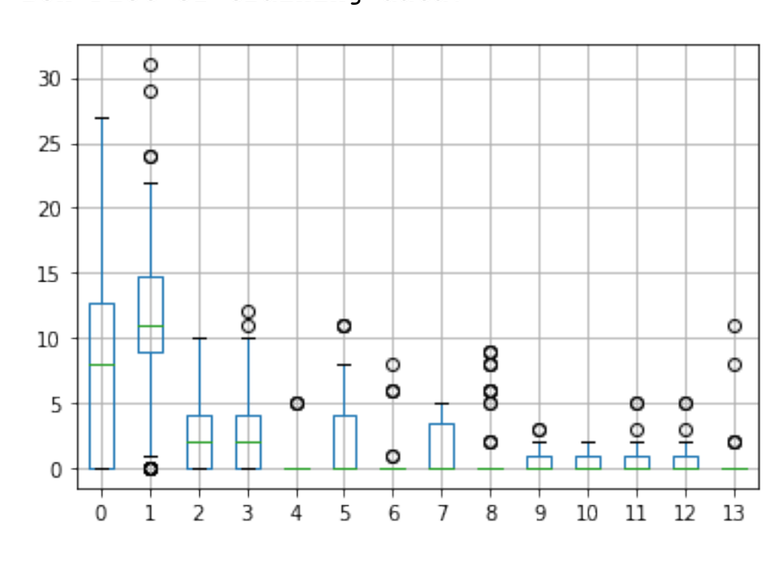
	0	1	2	3	4	5	6	7	8	9	...	30	31	32	33	34	35	36	37	38	39
0	8	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	12	10	1	1	0	0	0	4	6	0	...	0	0	0	0	0	0	0	0	0	0
3	21	10	4	4	0	1	1	0	5	0	...	0	0	0	0	0	0	0	0	0	0
4	27	12	3	3	0	8	0	5	0	2	...	0	0	0	0	0	0	0	0	0	0

5 rows × 40 columns

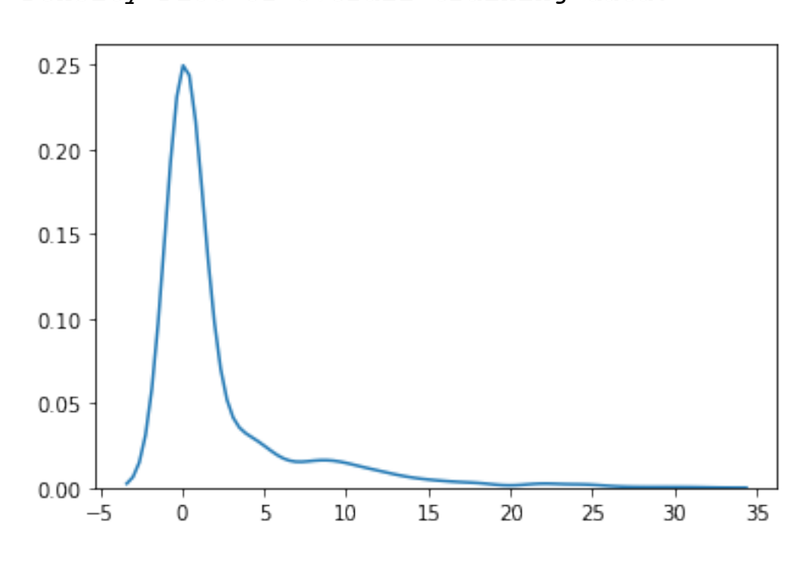
In [3]:
```
# Separate input features from output data
X_train_df = df_train.iloc[:, :14]
y_train_df = df_train.iloc[:, 14:]
X_test_df = df_test.iloc[:, :14]
y_test_df = df_test.iloc[:, 14:]
```

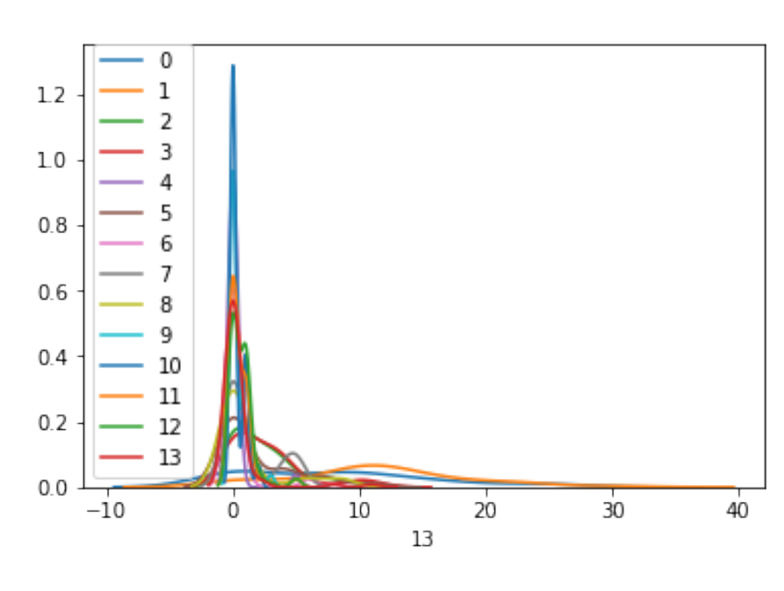
In [4]:
```
# Check whether data needs normalization
## Checking training data
### Box plot
print('Box Plot of training data: ')
X_train_df.boxplot()
plt.show()
print()

### Density plot
print('Density Plot of overall training data:')
sns.distplot(X_train_df, hist=False)
plt.show()


### Density plot of each feature in training data
for i in X_train_df:
    sns.distplot(X_train_df[i], hist = False, label = i)
plt.show()

## Checking testing data
### Box plot
print('Box Plot of testing data: ')
X_test_df.boxplot()
plt.show()
print()

### Density plot
print('Density Plot of overall testing data:')
sns.distplot(X_test_df, hist=False)
plt.show()

### Density plot of each feature in testing data
for i in X_test_df:
    sns.distplot(X_test_df[i], hist = False, label = i)
plt.show()
```

Box Plot of training data:

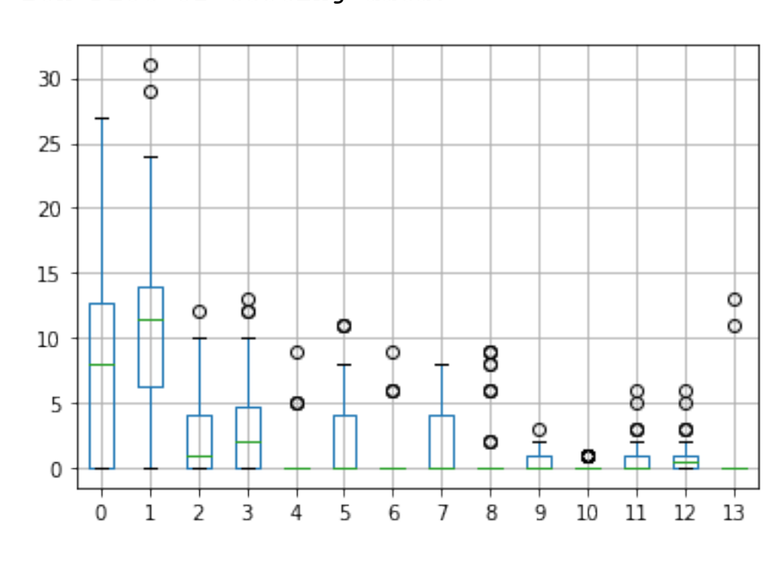


Density Plot of overall training data:

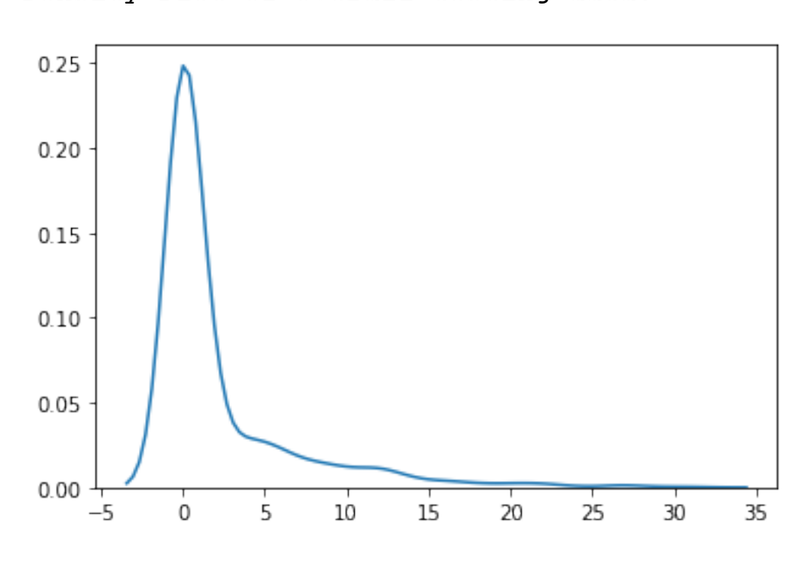


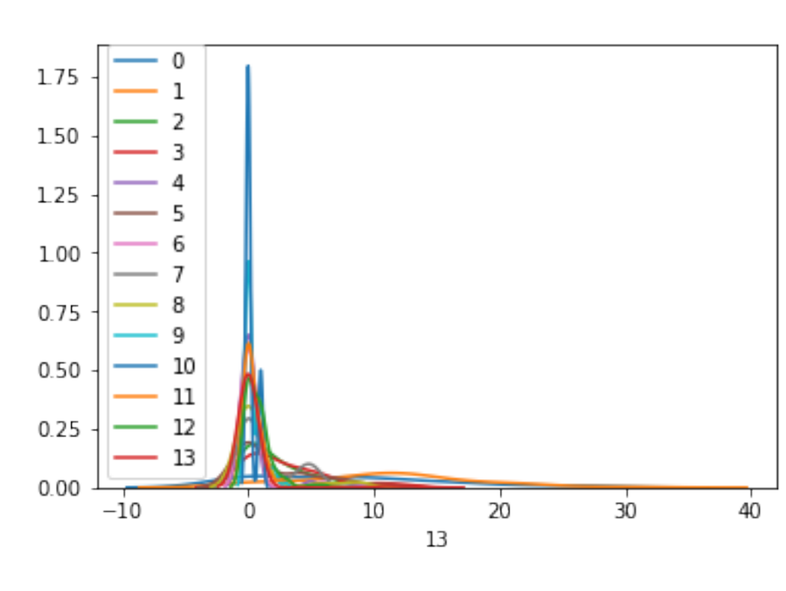


Box Plot of testing data:



Density Plot of overall testing data:



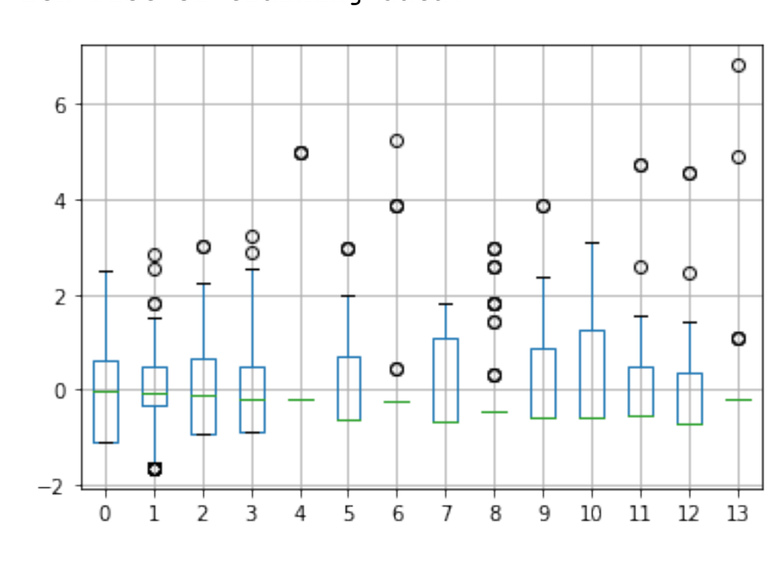


Varing mean and standard deviations in above plots indicate that we need to normalize data.

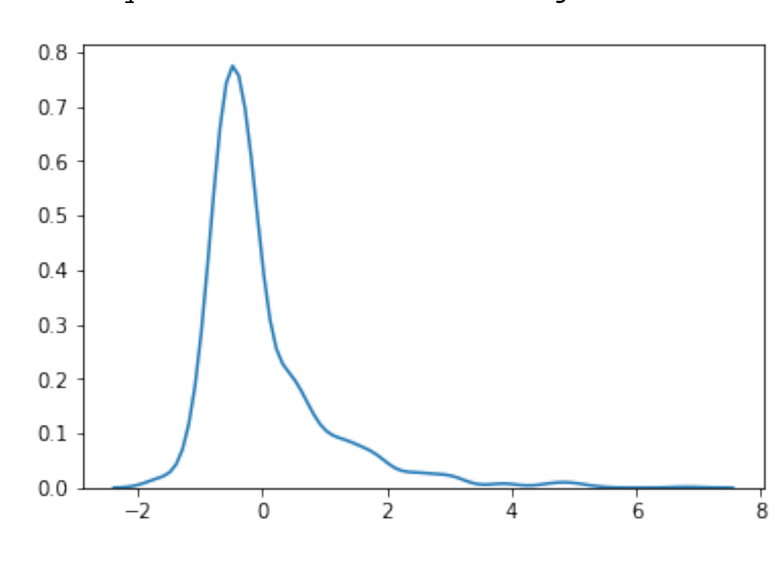
In [5]:
```
# Data Normalization
## Normalize training data
normalized_X_train_df=(X_train_df-X_train_df.mean())/X_train_df.std()

## Normalize testing data
normalized_X_test_df=(X_test_df-X_test_df.mean())/X_test_df.std()
```

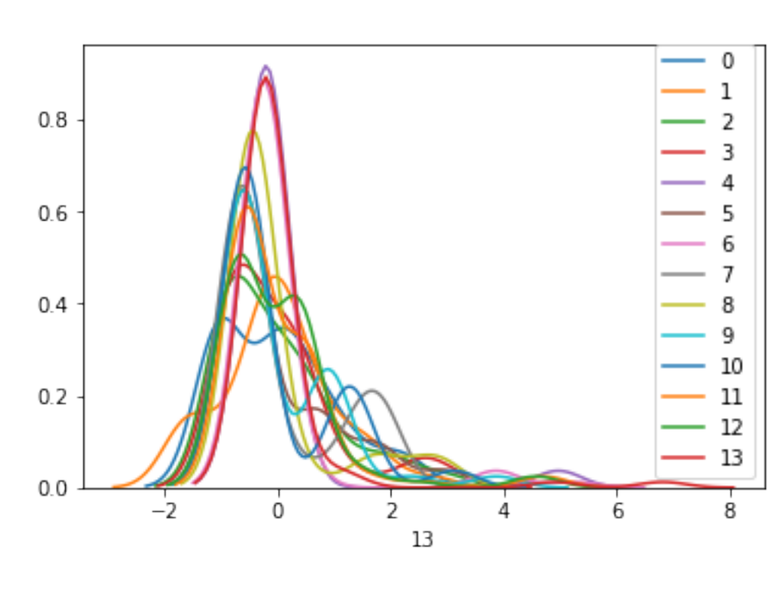
In [6]:
```
# Check normalized data
## Checking training data
### Box plot
print('Box Plot of training data: ')
normalized_X_train_df.boxplot()
plt.show()
print()

### Density plot
print('Density Plot of overall training data:')
sns.distplot(normalized_X_train_df, hist=False)
plt.show()


### Density plot of each feature in training data
for i in normalized_X_train_df:
    sns.distplot(normalized_X_train_df[i], hist = False, label = i)
plt.show()

## Checking testing data
### Box plot
print('Box Plot of testing data: ')
normalized_X_test_df.boxplot()
plt.show()
print()

### Density plot
print('Density Plot of overall testing data:')
sns.distplot(normalized_X_test_df, hist=False)
plt.show()

### Density plot of each feature in testing data
for i in normalized_X_test_df:
    sns.distplot(normalized_X_test_df[i], hist = False, label = i)
plt.show()
```

Box Plot of training data:

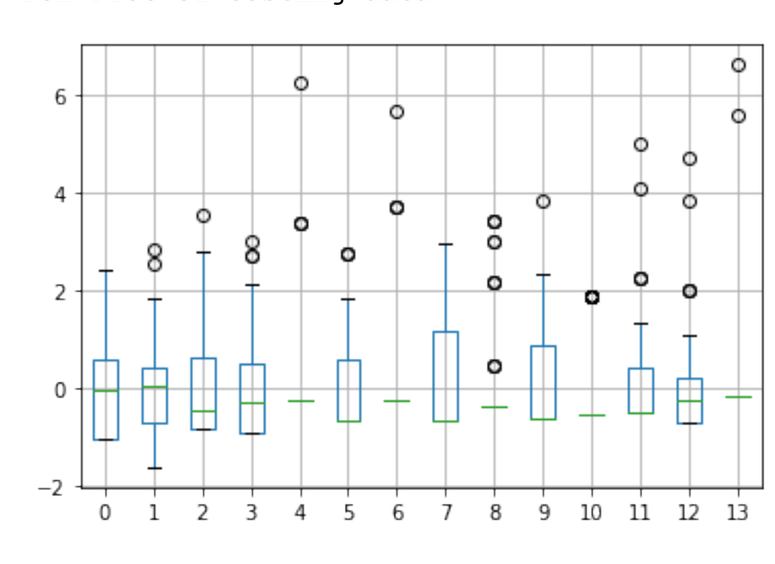


Density Plot of overall training data:

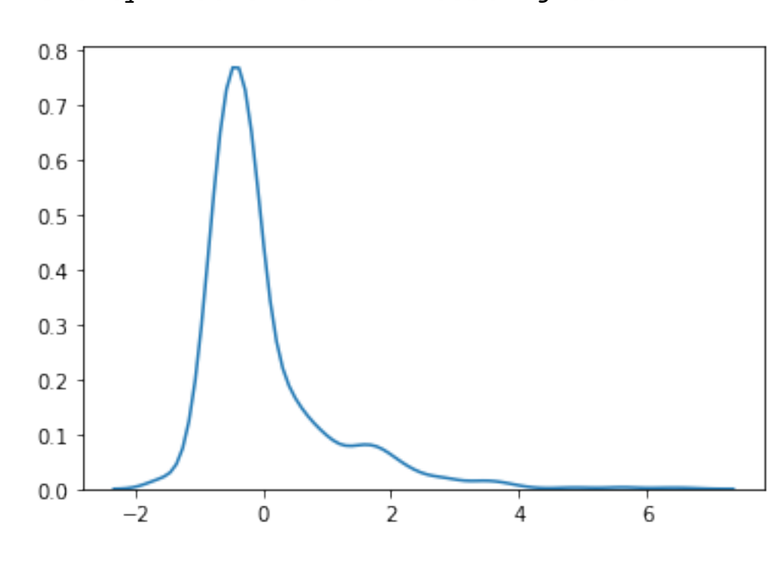


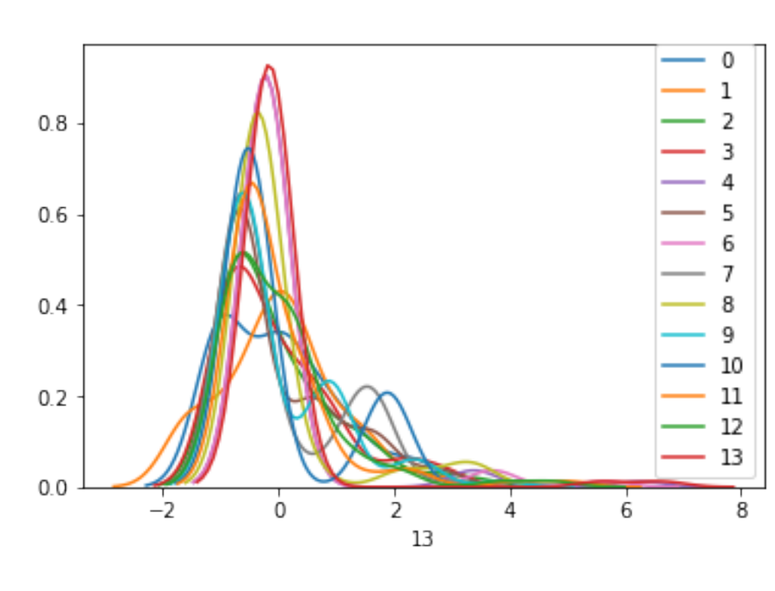


Box Plot of testing data:



Density Plot of overall testing data:





Since, we have data normalized, it can be converted to arrays to use futher for training and testing purposes.

In [7]:
```
# Convert dataframes into numpy array
X_train = X_train_df.values
y_train = y_train_df.values
X_test = X_test_df.values
y_test = y_test_df.values
```

In [8]:
```
# Preparing target labels

## Training data
count = 1
cls = 1
y_cls = []
for i in range (104):
    if count == 4:
        count = 1
        cls +=1
    else:
        y_cls.append(cls)
        count += 1

y_labeled = np.asarray(y_cls)
print(y_labeled)
```

```
[ 1  1  1  2  2  2  3  3  3  4  4  4  5  5  5  6  6  6  7  7  7  8  8  8
  9  9  9 10 10 10 11 11 11 12 12 12 13 13 13 14 14 14 15 15 15 16 16 16
 17 17 17 18 18 18 19 19 19 20 20 20 21 21 21 22 22 22 23 23 23 24 24 24
 25 25 25 26 26 26]
```

Build Model

Build Model and Tune Hyper-Parameters

In this section, we'll use grid search cross validation technique to find the best parameters from a given list. We'll also use stratified shuffle split technique to split our training and testing data.

In [9]:
```
# Parameters options
parameter_space = {'kernel': ['linear', 'rbf'], 'gamma': [0.001, 0.01, 0.1, 1], 'C': [0.001, 0.01, 0.1, 1, 10]}

# Cross-validation split
cv = StratifiedShuffleSplit(n_splits = 14, test_size=0.6, random_state = 1)

# Finding best parameters
clf = GridSearchCV(SVC(),parameter_space, cv = cv)
clf.fit(X_train, y_labeled)
print(clf.best_params_)
```

```
{'C': 1, 'gamma': 0.001, 'kernel': 'linear'}
```

Train Model - using optimized hyper-parameters

In [10]:
```
# Training the model using best parameters
clf.fit(X_train, y_labeled)
```

Out[10]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=14, random_state=1, test_size=0.6,
            train_size=None),
             error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 10],
                         'gamma': [0.001, 0.01, 0.1, 1],
                         'kernel': ['linear', 'rbf']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

Evaluate Model

In [11]:
```
# Predicting target labels on test data
y_pred = clf.predict(X_test)
y_test = y_labeled
```

In [12]:
```
# Best score on training dataset
clf.score(X_train, y_labeled)
```

Out[12]: 1.0

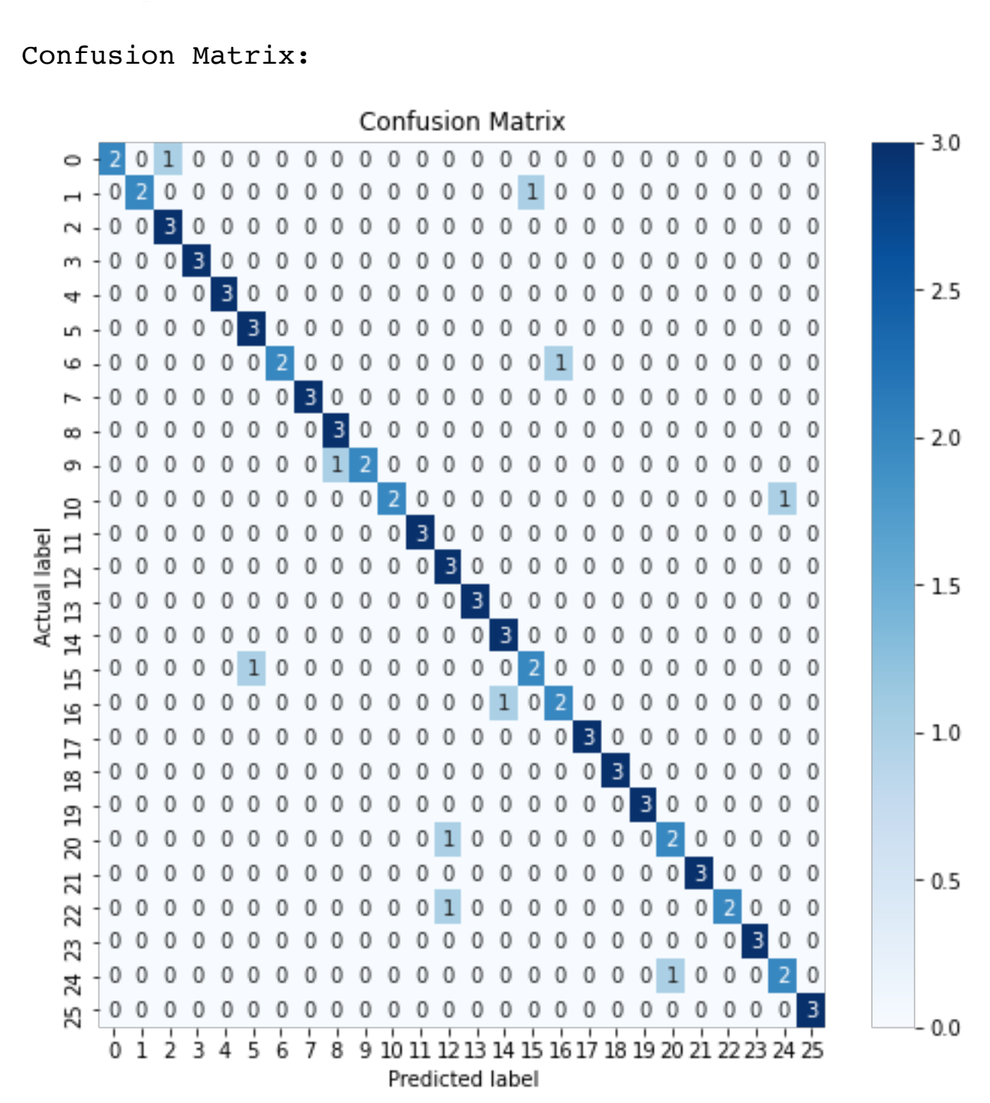
In [13]:
```
# Best score on testing dataset
clf.score(X_test,  y_labeled).round(2)
```

Out[13]: 0.87

In [14]:
```
# Print Model Accuracy Score
print ('Accuracy Score: {}'.format(accuracy_score(y_test, y_pred).round(2)))
print()

# Confusion Matrix
print('Confusion Matrix:')
results = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,8))
sns.heatmap(results, annot=True, fmt="d", cmap = 'Blues')
plt.title('Confusion Matrix')
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
print()

# Classification Report
print ('Classification Report: ')
print (classification_report(y_test, y_pred))
```

Accuracy Score: 0.87

Confusion Matrix:



Classification Report:

```
              precision    recall  f1-score   support

           1       1.00      0.67      0.80         3
           2       1.00      0.67      0.80         3
           3       0.75      1.00      0.86         3
           4       1.00      1.00      1.00         3
           5       1.00      1.00      1.00         3
           6       0.75      1.00      0.86         3
           7       1.00      0.67      0.80         3
           8       1.00      1.00      1.00         3
           9       0.75      1.00      0.86         3
          10       1.00      0.67      0.80         3
          11       1.00      0.67      0.80         3
          12       1.00      1.00      1.00         3
          13       0.60      1.00      0.75         3
          14       1.00      1.00      1.00         3
          15       0.75      1.00      0.86         3
          16       0.67      0.67      0.67         3
          17       0.67      0.67      0.67         3
          18       1.00      1.00      1.00         3
          19       1.00      1.00      1.00         3
          20       1.00      1.00      1.00         3
          21       0.67      0.67      0.67         3
          22       1.00      1.00      1.00         3
          23       1.00      0.67      0.80         3
          24       1.00      1.00      1.00         3
          25       0.67      0.67      0.67         3
          26       1.00      1.00      1.00         3

    accuracy                           0.87        78
   macro avg       0.89      0.87      0.87        78
weighted avg       0.89      0.87      0.87        78
```

Conclusion

Using Optimized SVM model, we are able to achieve an accuracy score of 87%, which, considering a highly sparse dataset, is quite a good score. There is still scope of further optimizing SVM model using randomized grid search option or using different cross validation technique.