DATA ANALYTICS (CS40003)

Group ID - 42
Akash Chandra (16MA20006)
Dibya Prakash Das (16MA20017)

Topic - 12

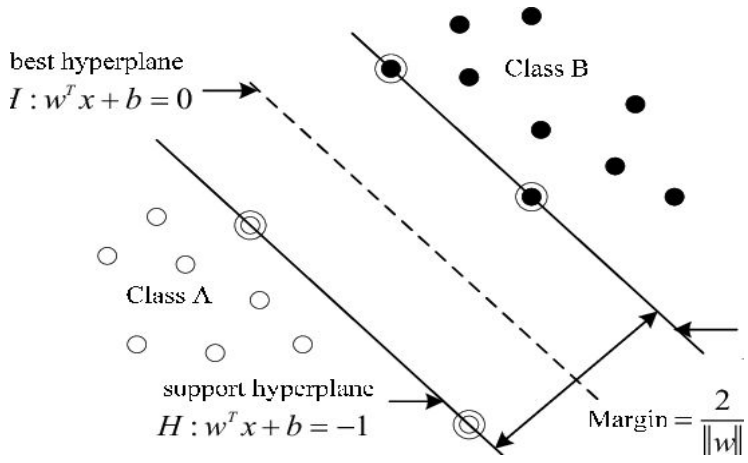
Error Based Classification for Non-Linear SVM

Introduction to Linear SVM :

Support Vector Machine(SVM) is a supervised machine learning algorithm for classification and regression analysis. For classification task, gIven a set of training examples, with each example class provided, SVM builds a model that assigns new or unseen examples to one category or other.

SVM is generally a non-probabilistic binary linear classifier, although methods such as **Platt Scaling** exists to use SVM in a probabilistic classification setting. And strategies such **One vs One(OvO)** and **One vs Rest(OvA)** exists to apply SVM for multiclass and multilabel classification problems.

A SVM model is a representation of points in space such that a clear gap exists separating both the categories and the objective of Linear SVM is to maximize this gap called **Margin**.



The two hyper-planes drawn with equation

$$H1 \ : \ w^T x \ + \ b \ = \ -1$$

$$H2 \ : \ w^T x \ + \ b \ = \ +1$$

where **w** refers to weight or coefficients and **b** refers to the intercept of hyperplane, represents two **support hyperplane** and any points in training set passing through hyperplane is referred to as **support vectors**.

Each tuple is denoted by $(X_i, \ Y_i)$ where $X_i \ = (\ X_{i1}, \ X_{i2}, \ X_{i3}, \ ..., \ X_{im})$ corresponds to the attribute set for the $i^{\ th}$ tuple (data in m-dimensional space) and Y_i [+, -] denotes the class labels for binary classification.

For a **positive data point (** $Y_i \ = \ \{+\}$ **)**, x_+ , the hyperplanes should be such that the projection of x_+ i.e. $w^T x_+ + \ b$ returns a value **greater than or equal to 1**.

$$w^T x_+ + \ b \ >= \ +1$$

For a **negative data point (** $Y_i \ = \ \{-\}$ **)**, x_- , the hyperplanes should be such that the projection of x_- i.e. $w^T x_- + \ b$ returns a value **less than or equal to -1**.

$$w^T x_- \ + \ b \ <= \ -1$$

We introduce another **variable** y_i , hence simplifying the constraints as

$$y_i(w^T x + \ b) \ >= \ +1$$

Now the optimization problem we are currently with is given as

$$Margin = \frac{2}{||w||}$$

The LinearSVM model objective is to **maximize the margin** between decision boundary(i.e. the two support hyperplane). Such a hyperplane obtained is called **Maximum Margin Hyperplane(MMH)**.

$$Minimize \ \frac{||w||^2}{2}$$

$$subject\ to\ y_i(w^T x_+ + b) >= +1$$

We can use **Gradient Descent** to minimize the said objective function, but is is not always guaranteed that we will reach a globally optimal solution, so here we will use the **Lagrangian** to solve for global minimum and this becomes a **convex optimization problem**.

$$L = \frac{1}{2}\|\vec{w}\|^2 - \sum_i^n \alpha_i \left[y_i(\vec{w}\cdot\vec{x} + b) - 1\right]$$

Constraints are given by **Karush-Kahn-Tucker (KKT)** Constraints which are as follows:

$$\frac{\partial L}{\partial w} = \vec{w} - \sum_i^n \alpha_i y_i x_i = 0$$

$$\boxed{\vec{w} = \sum_i^n \alpha_i y_i x_i}$$

$$\frac{\partial L}{\partial b} = -\sum_i^n \alpha_i y_i = 0$$

$$\boxed{\sum_i^n \alpha_i y_i = 0}$$

Substituting the above obtained values to the **Lagrangian**, we obtain the corresponding dual equation with the only variable as $\alpha\,(Lagrangian\ Multiplier)$.

$$L = \sum_i^n \alpha_i - \frac{1}{2}\sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j x_i \cdot x_j$$

In matrix form, the Lagrangian is given as

$$\begin{aligned} Maximize \quad & -(1/2)\alpha^T(X^T X)\alpha + 1^T x \\ subject\ to \quad & y^T\alpha = b \\ & \alpha_i <= c,\ i = 1,\ 2,\ 3, ...,\ n \end{aligned}$$

We use the library **CVXOPT** to solve convex optimization problem.

NOTE :: We haven't been taught in any course how to solve quadratic optimization problem, so we use the corresponding library.

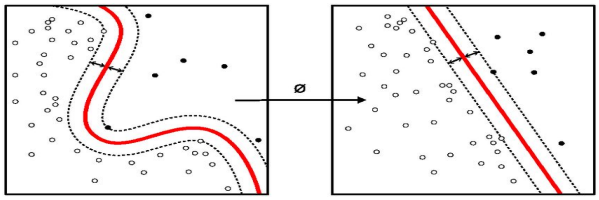
Linear SVM with Multiclass labels :

There are cases where we have to classify among more than 2 classes. It then becomes a problem of multi-class classification. We have two strategies we can follow to tackle this.

- **One-vs-One (OvO)** :- We train an SVM classifier for each pair of classes. And the class chosen by maximum number of SVMs is the correct one. The number of classifiers in this case is C_2^n. Time complexity increases exponentially with number of classes.
- **One-vs-All (OvA)** :- We have N binary classifiers for N-classes and each of these classifiers tells apart one class from the rest of the classes. Also called One-vs-Rest Classifiers. **OvA** requires **unanimity** among all SVMs: a data point would be classified under a certain class if and only if that class classifier accepted the data point and rest all classifiers rejected that data point.

Introduction to Non-Linear SVM :

In Non-Linear SVM, the trick is to transform the non-linearly separable data into higher dimension linearly-separable data. This transformation is referred to as **non-linear mapping** Or **attribute transformation**.



KERNEL TRICK

The convex objective function is of the form

$$Classifier: \ \delta(X) = \sum_{i=1}^{n} \lambda_i . Y_i . K(X_i, X) + b$$

$$Learning: \ maximize \sum_{i=1} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i . \lambda_j Y_i Y_j . K(X_i . X_j)$$

$$Subject\ to\ \lambda_i \geq 0\ and \sum_{} \lambda_i . Y_i = 0$$

In matrix form, the Lagrangian is given as

$$Maximize \quad -(1/2)\alpha^T Q\alpha + 1^T x$$

$$where, \ Q_{ij} \equiv y_i y_j K(x_i, x_j)$$

$$subject\ to \quad y^T \alpha = b$$

$$\alpha_i <= c, \ i = 1, 2, 3, ..., n$$

We use the library **CVXOPT** to solve convex optimization problem.

NOTE :: We haven't been taught in any course how to solve quadratic optimization problem, so we use the corresponding library.

Our Approach towards solution :

STEP-Ⅱ (Visualizing the dataset and applying appropriate scaling techniques along with Label Encoding of the Target variable)

IMPORTANT CONCLUSIONS FROM DATASET ***(GLASS.CSV)***

- Total no of data points - **214**
- There is no missing attribute in the dataset
- FIrst five rows of data are as **(9 attributes and 1 target variable)**

	0	1	2	3	4	5	6	7	8	9	10
0	1	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	2	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	3	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	4	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	5	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

- Number of target-labels is **SIX** with counts as

1	70
2	76
3	17
5	13
6	9
7	29

LABEL-ENCODING

Transformation of target-labels using custom build **LabelEncoder class**.

old_class_name	new_class_name
1	0
2	1
3	2
5	3
6	4
7	5

SCALING-ATTRIBUTES

```
data.describe()
```

	1	2	3	4	5	6	7	8	9
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000
mean	1.518365	0.402684	2.684533	1.444907	0.507310	0.497056	0.382802	0.175047	0.057009
std	0.003037	0.122798	1.442408	0.499270	0.138312	0.652192	1.526170	0.497219	0.097439
min	1.511150	0.000000	0.000000	0.290000	0.000000	0.000000	-3.399464	0.000000	0.000000
25%	1.516523	0.327444	2.115000	1.190000	0.441071	0.122500	-0.386059	0.000000	0.000000
50%	1.517680	0.386466	3.480000	1.360000	0.532143	0.555000	0.000000	0.000000	0.000000
75%	1.519157	0.465414	3.600000	1.630000	0.585268	0.610000	0.613941	0.000000	0.100000
max	1.533930	1.000000	4.490000	3.500000	1.000000	6.210000	8.139410	3.150000	0.510000

For **attribute 2 and 5**, we apply **MinMaxScaler**, to scale the columns in the feature range of **[0, 1]**.
It is done by following equation:

$$X_{scaled} = scale * X + min - X_{min}$$
$$where\ scale\ = \frac{(max - min)}{(X_{max} - X_{min})}$$

Code Snippet

```
class MinMaxScaler:
    def __init__(self, feature_range=(0, 1)):
        self.feature_range = feature_range
    def fit(self, X):
        data_min = np.nanmin(X, axis=0)
        data_max = np.nanmax(X, axis=0)
        self.data_range = (data_max - data_min)
        self.scale_ = (self.feature_range[1] - self.feature_range[0]) / data_range
        self.min_ = self.feature_range[0] - data_min * self.scale_
        self.data_min_ = data_min
        self.data_max_ = data_max
    def transform(self, X):
        X = np.asarray(X, dtype=np.float64)
        X *= self.scale_
        X += self.min_
        return X
```

For **attribute 7**, Robust Scaling **(RobustScaler)** was applied, as it is robust to **outliers and noise**.
It is done by the following equation:

$$X_{norm} = \frac{X - Q_1(x)}{Q_3(x) - Q_1(x)}$$

Code Snippet

```
class RobustScaler:
    def __init__(self, with_centering=True, with_scaling=True,
quantile_range=(25.0, 75.0)):
        self.with_centering = with_centering
        self.with_scaling = with_scaling
        self.quantile_range = quantile_range
    def fit(self, X):
        q_min, q_max = self.quantile_range
        self.center_ = np.nanmedian(X, axis=0) if self.with_centering else None
        if self.with_scaling:
            quantiles = []
            for feature_idx in range(X.shape[1]):
                column_data = X[:, feature_idx]
                quantiles.append(np.nanpercentile(column_data, self.quantile_range))
            quantiles = np.transpose(quantiles)
            self.scale_ = quantiles[1] - quantiles[0]
        else:
            self.scale_ = None
        return self
    def transform(self, X):
        X = np.asarray(X, dtype=np.float64)
        if self.with_centering:
            X -= self.center_
        if self.with_scaling:
            X /= self.scale_
        return X
```

Now the updated descriptive statistics after scaling is

```
data.describe()
```

	1	2	3	4	5	6	7	8	9
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000
mean	1.518365	0.402684	2.684533	1.444907	0.507310	0.497056	0.382802	0.175047	0.057009
std	0.003037	0.122798	1.442408	0.499270	0.138312	0.652192	1.526170	0.497219	0.097439
min	1.511150	0.000000	0.000000	0.290000	0.000000	0.000000	-3.399464	0.000000	0.000000
25%	1.516523	0.327444	2.115000	1.190000	0.441071	0.122500	-0.386059	0.000000	0.000000
50%	1.517680	0.386466	3.480000	1.360000	0.532143	0.555000	0.000000	0.000000	0.000000
75%	1.519157	0.465414	3.600000	1.630000	0.585268	0.610000	0.613941	0.000000	0.100000
max	1.533930	1.000000	4.490000	3.500000	1.000000	6.210000	8.139410	3.150000	0.510000

STEP-II (Splitting the data into train and test set and applying Stratified 10-fold Cross Validation)

SPLITTING THE DATASET INTO TRAIN AND TEST

The data is split into train and test in the ratio $9:1$, where $Num_{train} = 0.9 * len(data)$ is the number of data points in the training set and $Num_{test} = 0.1 * len(data)$ is the number of data points in the test set.

```
X, y, X_test, y_test = train_test_split(X, y, test_size=0.1, shuffle=True)
```

Code Snippet

```
def train_test_split(X, y, test_size=0.2, shuffle=True):
    n_samples = X.shape[0]
    if shuffle:
        indices = np.arange(n_samples)
        random.shuffle(indices)
        X, y = X[indices], y[indices]
    test_samples = int(n_samples * test_size)
    return X[ :-test_samples], y[ :-test_samples], X[-test_samples: ],
y[-test_samples: ]
```

UTILITY CODE FOR PERFORMING STRATIFIED K-FOLD CROSS VALIDATION

Code Snippet

```
class StratifiedKFold(BaseKFold):
    def __init__(self, n_splits=10, shuffle=True):
        super().__init__(n_splits, shuffle)

    def split(self, X, y):
        for train, test in super().split(X, y):
            yield train, test

    def _iter_test_masks(self, X, y):
        test_folds = self._make_test_folds(X, y)
        for i in range(self.n_splits):
            yield test_folds == i

    def _make_test_folds(self, X, y):
        _, y_idx, y_inv = np.unique(y, return_index=True, return_inverse=True)
        _, class_perm = np.unique(y_idx, return_inverse=True)
        y_encoded = class_perm[y_inv]
        n_classes = len(y_idx)
        y_counts = np.bincount(y_encoded)
        min_groups = np.min(y_counts)
        y_order = np.sort(y_encoded)
        allocation = np.asarray([np.bincount(y_order[i::self.n_splits],
minlength=n_classes) for i in range(self.n_splits)])
        test_folds = np.empty(len(y), dtype='i')
        for k in range(n_classes):
            folds_for_class = np.arange(self.n_splits).repeat(allocation[:, k])
            if self.shuffle:
                random.shuffle(folds_for_class)
            test_folds[y_encoded == k] = folds_for_class
        return test_folds
```

Code Snippet

```
for i, (train_index, val_index) in
enumerate(StratifiedKFold(n_splits=10).split(X, y)):
    X_train, X_val, y_train, y_val = X[train_index], X[val_index],
y[train_index], y[val_index]
    model = SVC(C=0.5, kernel='poly', coef0=1.0, degree=5, gamma='scale')
    classifiers = model.fit(X_train, y_train)
    y_pred = model.predict(X_val)
    y_train_pred = model.predict(X_train)
    print("Train Accuracy :: ", accuracy_score(y_train, y_train_pred))
    print(classification_report(y_val, y_pred))
```

STEP-エエエ (Validating the Non-Linear SVM model for different kernels)

VARIOUS KERNELS USED FOR NON-LINEAR SVM

- **Linear Kernel -** It is the simplest type of kernel function and prediction of new inputs take place using the dot product between the $input(x)$ and $support\ vector\ (x_i)$.

$$K(x_i, x_j) = < x_i, x_j >$$

Code Snippet

```
def linear_kernel(x, y, **kwargs):
    return np.dot(x, y)
```

- **Polynomial Kernel -** It is a non-stationary kernel. Polynomial kernels are well-suited for problems where all the training data is normalized.

$$K(x_i, x_j) = (\gamma * < x_i, x_j > + r)^d$$

Code Snippet

```
def polynomial_kernel(x, y, **kwargs):
    degree = kwargs['degree']
    gamma = kwargs['gamma']
    coef0 = kwargs['coef0']
    return (coef0 + gamma * np.dot(x, y)) ** degree
```

- **Gaussian Radial Basis Function (RBF) -** When training an SVM with the *Radial Basis Function Kernel (RBF)*, two parameters C and γ are important. Parameter C common to all the SVM kernels, is a tradeoff between misclassification of training examples against simplicity of the decision surface.
  γ decides how much influence a single training example has, the larger values of γ implies closer data points must be affected.

$$K(x_i, x_j) = exp(-\gamma * \left\|x_i - x_j\right\|^2)$$

Code Snippet

```
def gaussian_kernel(x, y, **kwargs):
    gamma = kwargs['gamma']
    return np.exp(-gamma * (np.linalg.norm(x-y)**2))
```

UTILITY CODE TO SOLVE THE CONVEX OPTIMIZATION PROBLEM

The convex optimization problem is in the form of

$$\begin{aligned} &Minimize && (1/2)x^T Px + q^T x \\ &subject\ to && Gx <= h \\ & && Ax = b \end{aligned}$$

Code Snippet

```
K = np.zeros(shape = (n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i][j] = self.kernel_function(X[i], X[j], **self.kwargs)
P = cvxopt.matrix(np.outer(y,y) * K)
q = cvxopt.matrix(np.ones(n_samples) * -1)
A = cvxopt.matrix(y, (1, n_samples), 'd')
b = cvxopt.matrix(0.0)
tmp1 = np.diag(np.ones(n_samples) * -1)
tmp2 = np.identity(n_samples)
G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
tmp1 = np.zeros(n_samples)
tmp2 = np.ones(n_samples) * self.C
```

```
h = cvxopt.matrix(np.hstack((tmp1, tmp2)))
# solve QP problem
solution = cvxopt.solvers.qp(P, q, G, h, A, b)
```

Results :

F1-SCORE USING STRATIFIED 10-FOLD CROSS VALIDATION ON TRAIN DATA

- **Linear Kernel (** $C = 1.0$ **)**

  $accuracy$: 61.078947 (+/− 8.997114)

  $macro\ average\ f1-score$: 45.147540 (+/− 10.434229)

  $weighted\ average\ f1-score$: 56.677573 (+/− 9.008458)

- **Polynomial Kernel (** $C = 0.5,\ coef0 = 1.0,\ degree = 5,\ gamma = 'scale'$ **)**

  $accuracy$: 64.342105 (+/− 9.212970)

  $macro\ average\ f1-score$: 54.161567 (+/− 10.905636)

  $weighted\ average\ f1-score$: 62.116334 (+/− 9.257801)

- **RBF Kernel (** $C = 0.5,\ gamma = 'scale'$ **)**

  $accuracy$: 62.789474 (+/− 11.450967)

  $macro\ average\ f1-score$: 45.782323 (+/− 12.266286)

  $weighted\ average\ f1-score$: 58.362990 (+/− 11.140270)

CONFUSION MATRIX, CLASSIFICATION REPORT FOR TEST DATA

- **Linear Kernel (** $C = 1.0$ **)**

Confusion Matrix

```
[[8 5 0 0 0 0]
 [2 1 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 0 1 0 0]
 [0 2 0 0 0 0]
 [0 0 0 1 0 0]]
```

Classification Report

	precision	recall	f1-score	support
0	0.80	0.62	0.70	13
1	0.11	0.33	0.17	3
2	0.00	0.00	0.00	1
3	0.50	1.00	0.67	1
4	0.00	0.00	0.00	2
5	0.00	0.00	0.00	1
accuracy			0.48	21
macro avg	0.24	0.32	0.25	21
weighted avg	0.53	0.48	0.49	21

- **Polynomial Kernel (** $C = 0.5,\ coef0 = 1.0,\ degree = 5,\ gamma = 'scale'$ **)**

  Confusion Matrix

```
[[12  1  0  0  0  0]
 [ 1  2  0  0  0  0]
 [ 1  0  0  0  0  0]
 [ 0  0  0  1  0  0]
 [ 0  0  0  0  2  0]
 [ 0  0  0  1  0  0]]
```

Classification Report

	precision	recall	f1-score	support
0	0.86	0.92	0.89	13
1	0.67	0.67	0.67	3
2	0.00	0.00	0.00	1
3	0.50	1.00	0.67	1
4	1.00	1.00	1.00	2
5	0.00	0.00	0.00	1
accuracy			0.81	21
macro avg	0.50	0.60	0.54	21
weighted avg	0.74	0.81	0.77	21

- **RBF Kernel (** $C = 0.5,\ gamma = 'scale'$ **)**

  Confusion Matrix

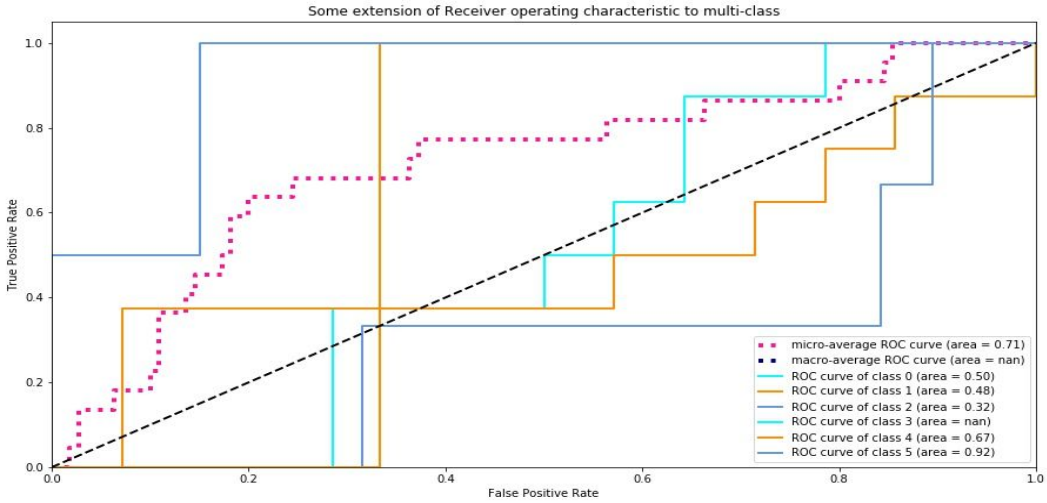
```
[[9 4 0 0 0 0]
 [1 2 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 0 1 0 0]
 [1 0 0 0 1 0]
 [0 0 0 1 0 0]]
```

  Classification Report

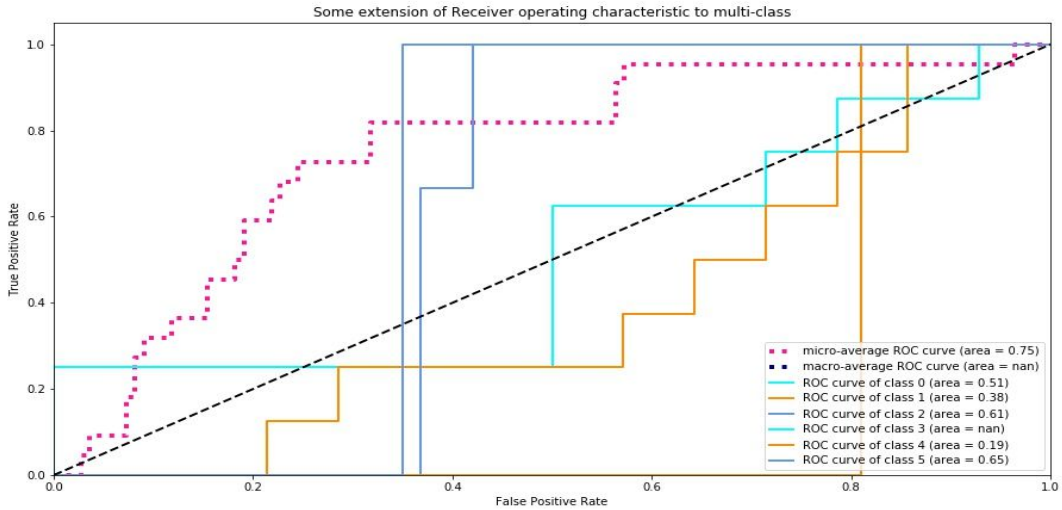
	precision	recall	f1-score	support
0	0.82	0.69	0.75	13
1	0.29	0.67	0.40	3
2	0.00	0.00	0.00	1
3	0.50	1.00	0.67	1
4	1.00	0.50	0.67	2
5	0.00	0.00	0.00	1
accuracy			0.62	21
macro avg	0.43	0.48	0.41	21
weighted avg	0.67	0.62	0.62	21

ROC, AUC for test data

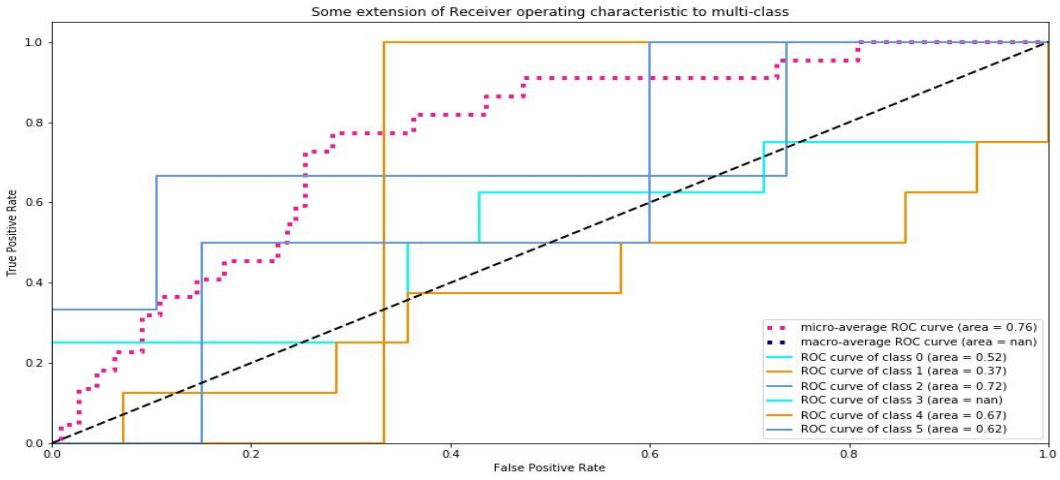
- **Linear Kernel (** $C = 1.0$ **)**



- **Polynomial Kernel (** $C = 0.5,\ coef0 = 1.0,\ degree = 5,\ gamma = 'scale'$ **)**



- **RBF Kernel (** $C = 0.5,\ gamma = 'scale'$ **)**



Conclusions :

For each kernel, using Stratified 10-fold cross validation best values of hyperparameters such as $C,\ \gamma,\ degree,\ coef0$ were found and later $f1-score,\ confusion\ matrix$ and $classification\ report$ were found on the test data which was separated initially and consisted of $0.1\ *\ n_{samples}$ data points.

After that from ROC curve, again on test data, we can see that RBF-kernel and Poly-kernel have similar performance thus get good performance with the two kernels, with the calculated hyperparameters.