CLASS: B.E. E &TC **SUBJECT: ML**
EXPT. NO.: 6 **DATE:**
ROLL NO.: 42428

TITLE: Implement SVM classifier for classification of data into two classes

---

CO 2: Design and Implement machine learning solution to address specified problems of classification, regression and clustering. Analyze the effects of dimensionality reduction using principal component analysis, factor analysis, multidimensional scaling. Evaluate and interpret the results of algorithms.

CO 4: Carry out experiments as an individual and in a team, comprehend and write a laboratory record and draw conclusions at a technical level.

AIM:
To implement:
1. Binary Classification using SVM

SOFTWARES REQUIRED: MATLAB 7.0 or Python with Skleran Library

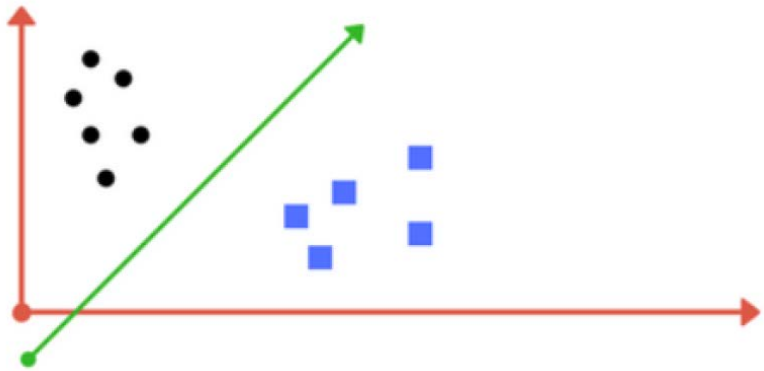
THEORY:

Support Vector Machine:
A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labelled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two- dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side. SVMs are helpful in text and hypertext categorization. Classification of images can also be performed using SVMs, Hand-written characters can be recognized using SVM.

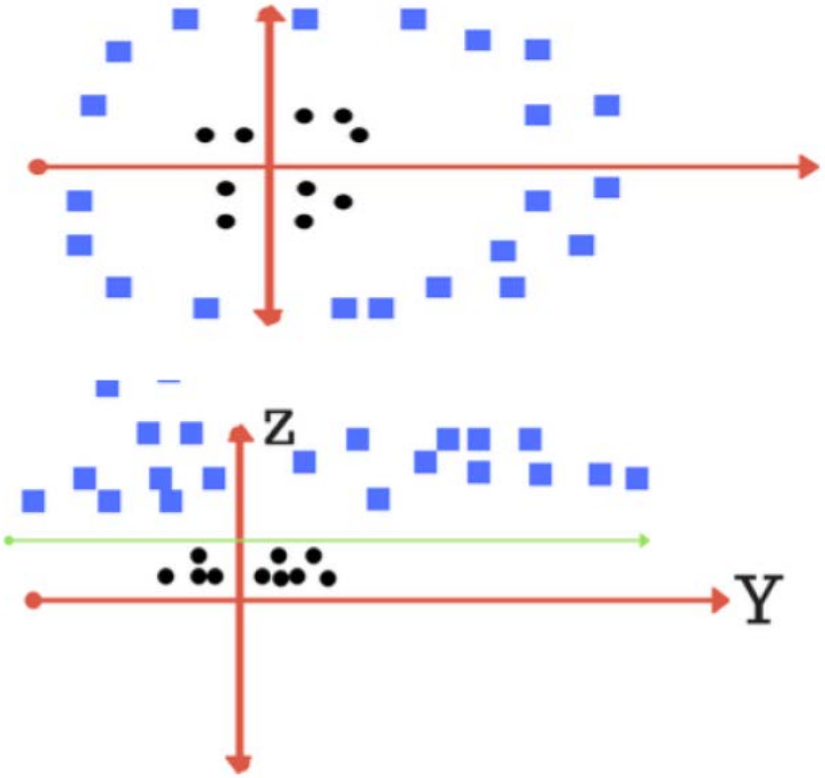
Support vector machine is a discriminative (data of different types) classifier. Support vector machines (also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression.

SVM performs separation of classes. Any point on left of separation line falls in class A category and on right falls in class B category. Separation line should be at a maximum distance from supporting vectors. This helps in increasing accuracy of system.

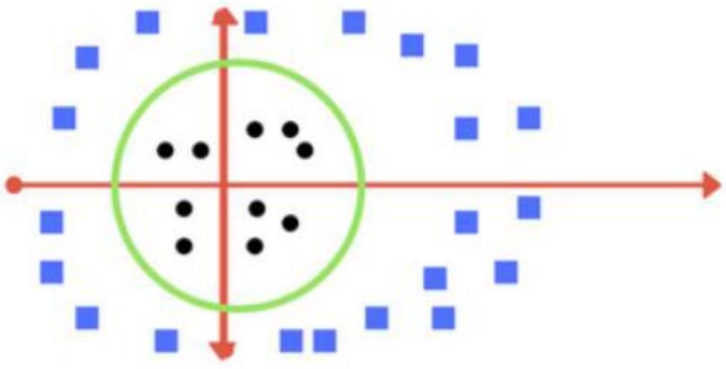
SVM differs from the other classification algorithms in the way that it chooses the decision boundary that maximizes the distance from the nearest data points of all the classes. An SVM doesn't merely find a decision boundary; it finds the most optimal decision boundary.



If there is no line that can separate the two classes in this x-y plane, we apply transformation and add one more dimension as we call it z-axis. Let's assume value of points on z plane, $w = x^2 + y^2$. In this case we can manipulate it as distance of point from z-origin. Now if we plot in z-axis, a clear separation is visible and a line can be drawn.



When we transform back this line to original plane, it maps to circular boundary. These transformations are called **kernels.** The sklearn library's SVM implementation provides it inbuilt.



Transforming back to x-y plane, a line transforms to circle

Kernel:

The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra. This is where the kernel plays role.

For linear kernel the equation for prediction for a new input using the dot product between the input (x) and each support vector (xi) is calculated as follows:

f(x) = B(0) + sum(ai * (x,xi))

This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients B0 and ai (for each input) must be estimated from the training data by the learning algorithm.

The polynomial kernel can be written as K(x,xi) = 1 + sum(x * xi)d

and exponential as

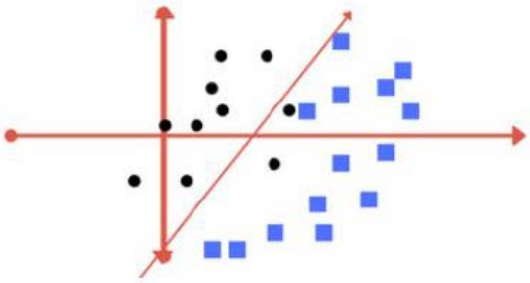
K(x,xi) = exp(-gamma * sum((x—xi²))

Regularization:

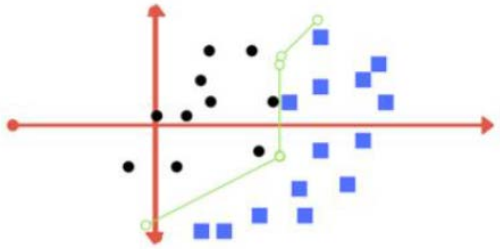
The Regularization parameter (often termed as C parameter in python's sklearn library) tells the SVM optimization how much you want to avoid misclassifying each training example.

For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly.

Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.



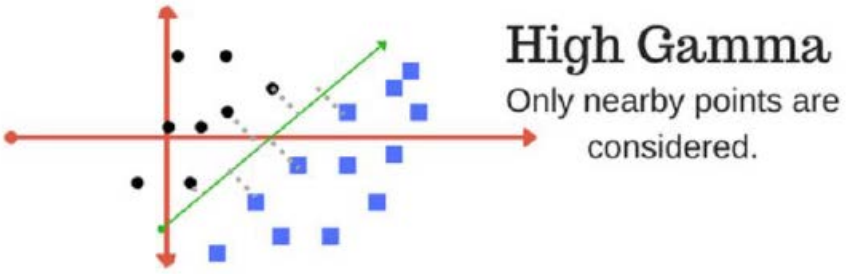
Low regularization



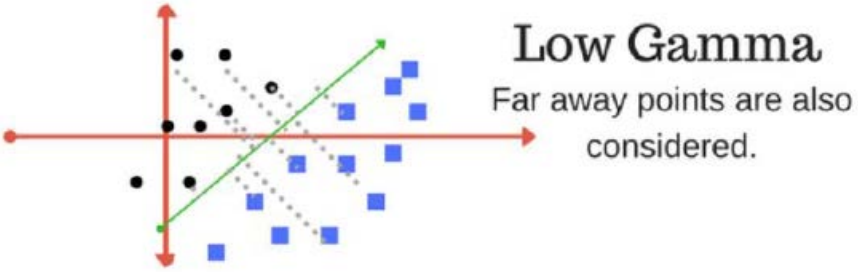
High regularization

Gamma:

The gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’. In other words, with low gamma, points far away from plausible separation line are considered in calculation for the separation line. Whereas high gamma means the points close to plausible line are considered in calculation.



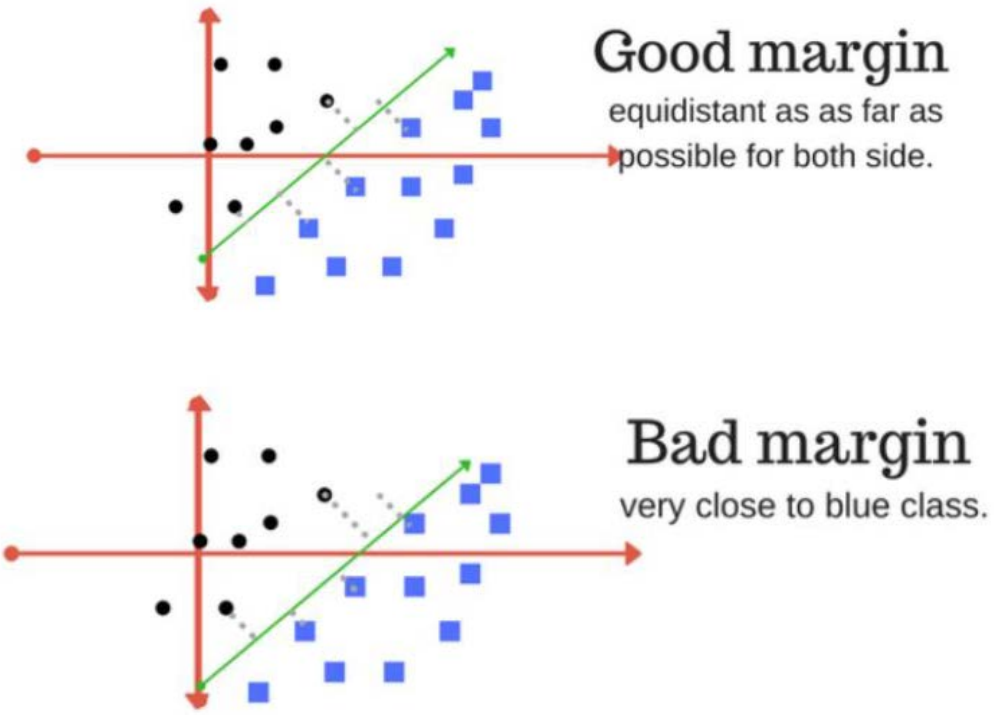
High Gamma



Low Gamma

Margin:

And finally last but very important characteristic of SVM classifier. SVM to core tries to achieve a good margin.
A margin is a separation of line to the closest class points.
A good margin is one where this separation is larger for both the classes. A good margin allows the points to be in their respective classes without crossing to other class.



CONCLUSION:

It can directly handle ready-to-use spectrograms of volcanic tremor as well as compute them from the original time series. It carries out the training of the SVM classifier with different parameters and kernels, and allows evaluating the performance of the classifier either on an independent test set or using a leave-one-out procedure. Classification results are presented by means of a graphical user interface in the form of overall classification accuracy, confusion

matrices, and with the help of graphs showing the a priori and assigned class for each considered pattern.

REFERENCES:

i. Kevin Murphy, Machine Learning: A Probabilistic Perspective, MIT Press, 2012.

ii. Trevor Hastie, Robert Tibshirani, Jerome Friedman, ―The Elements of Statistical Learning‖, Springer 2009.

iii. Phil Kim, ―MATLAB Deep Learning: With Machine Learning, Neural Networks and Artificial Intelligence‖,a Press 2017.

iv. Ethem Alpaydın ―Introduction to Machine Learning‖ Second Edition The MIT Press 2010.

(Course Teacher)

Code :

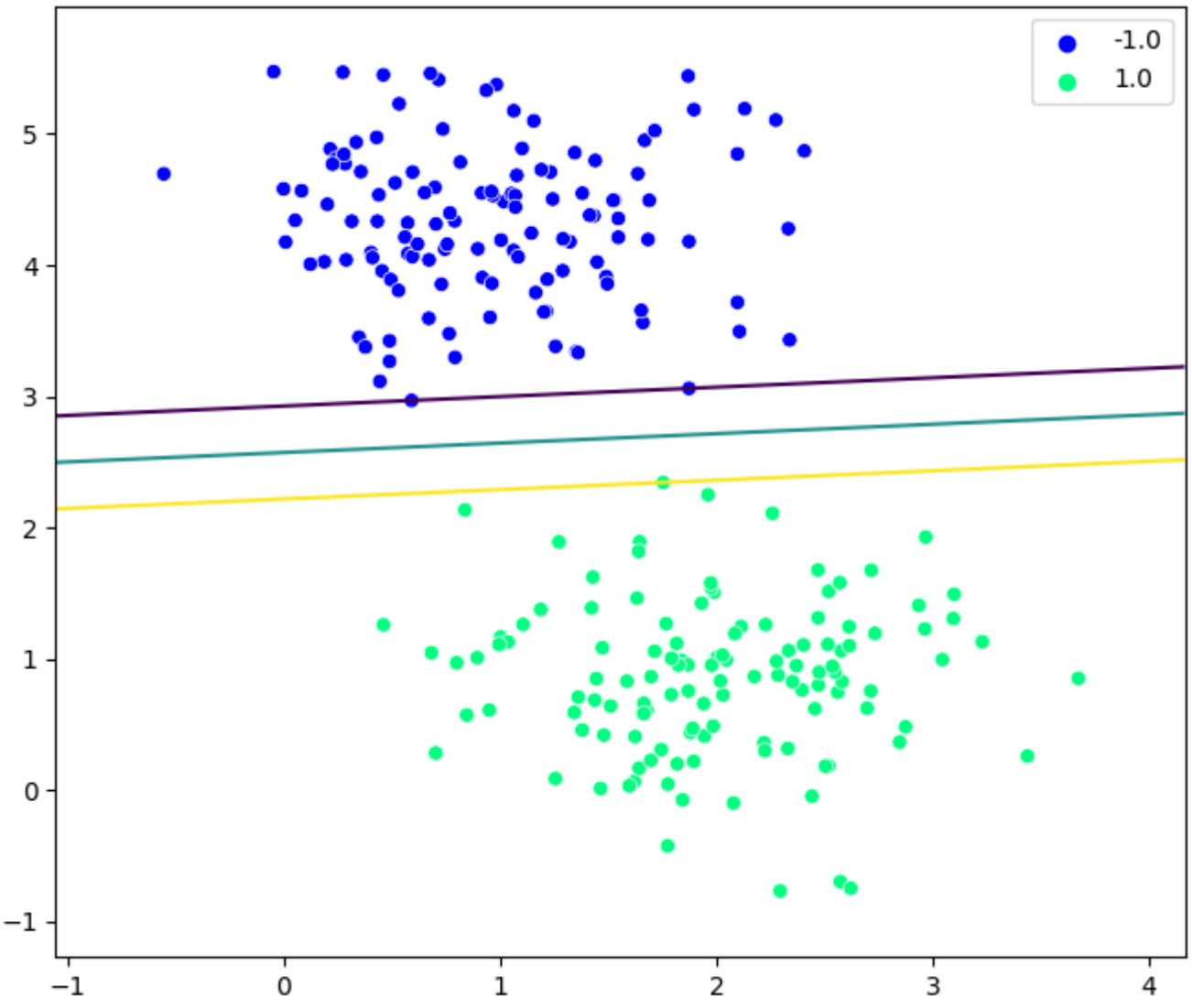
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_blobs
from cvxopt import matrix, solvers


def kernel(name, x, y):
    if name == 'linear':
        return np.dot(x, y.T)
    if name == 'poly':
        return (1 + np.dot(x, y.T)) ** 3


class SVM:
    def __init__(self):
        pass

    def fit(self, X, y, kernel_name='linear'):
        # Store for later use
        self.X = X
        self.y = y
        self.m = X.shape[0]
        self.kernel_name = kernel_name

        # Create optimization problem matrices
        P = np.empty((self.m, self.m))
        for i in range(self.m):
            for j in range(self.m):
                P[i, j] = y[i] * y[j] * kernel(kernel_name, X[i], X[j])
        q = -np.ones((self.m, 1))
        G = -np.eye(self.m)
        h = np.zeros((self.m, 1))
        A = y.reshape((1, self.m))
        b = np.zeros((1, 1))

        # Convert to CVXOPT matrix format
        P = matrix(P)
        q = matrix(q)
        G = matrix(G)
        h = matrix(h)
```

```
        A = matrix(A.astype('double'))
        b = matrix(b)

        # Solve Optimization Problem
        sol = solvers.qp(P, q, G, h, A, b)
        self.lambdas = np.array(sol['x']).reshape(self.m)

        # Calculate b
        SV = np.where(self.lambdas > 1e-4)[0][0]
        self.b = y[SV] - sum(self.lambdas * y * kernel(kernel_name, X, X[SV]))

    # Plot scatterplot of data and contour plot of SVM
    def plot(self):
        x_min = min(self.X[:, 0]) - 0.5
        x_max = max(self.X[:, 0]) + 0.5
        y_min = min(self.X[:, 1]) - 0.5
        y_max = max(self.X[:, 1]) + 0.5
        step = 0.02
        xx, yy = np.meshgrid(np.arange(x_min, x_max, step), np.arange(y_min, y_max, step))
        d = np.concatenate((xx.ravel().reshape(-1, 1), yy.ravel().reshape(-1, 1)), axis=1)

        Z = self.b + np.sum(
            self.lambdas.reshape((self.m, 1)) * self.y.reshape((self.m, 1)) * kernel(self.
kernel_name, self.X, d),
            axis=0)
        Z = Z.reshape(xx.shape)

        fig, ax = plt.subplots()
        sns.scatterplot(x=self.X[:, 0], y=self.X[:, 1], hue=self.y, ax=ax, palette='winter
')
        ax.contour(xx, yy, Z, levels=[-1, 0, 1])
        plt.show()

    # Predict class of given data point
    def predict(self, u):
        if self.b + sum(self.lambdas * self.y * kernel(self.kernel_name, self.X, u)) >= 0:
            return 1
        else:
            return -1


svm = SVM()
```

```
# Linearly Separable Example
x,y = make_blobs(n_samples=250,centers=2,random_state=0,cluster_std=0.60)
y[y==0]=-1
tmp=np.ones(len(x))
y=tmp*y
svm.fit(x, y)
svm.plot()
```

Output :

```
     pcost       dcost       gap    pres   dres
 0: -2.3987e+01 -4.4897e+01  8e+02  3e+01  2e+00
 1: -3.3503e+01 -2.2158e+01  3e+02  9e+00  6e-01
 2: -5.3318e+01 -3.3180e+01  3e+02  9e+00  6e-01
 3: -1.1170e+02 -5.7507e+01  3e+02  7e+00  5e-01
 4: -7.9798e+01 -2.6454e+01  2e+02  4e+00  3e-01
 5: -1.2608e+01 -5.0784e+00  2e+01  4e-01  3e-02
 6: -3.5365e+00 -4.2980e+00  8e-01  4e-15  3e-14
 7: -3.9998e+00 -4.0137e+00  1e-02  9e-16  1e-14
 8: -4.0093e+00 -4.0094e+00  1e-04  7e-16  1e-14
 9: -4.0094e+00 -4.0094e+00  1e-06  3e-15  1e-14
Optimal solution found.
```


-1.0
1.0
5
4
3
2
1
0
−1
−1
0
1
2
3
4