Lab 2: Medical Signal Segmentation and Classification

Introduction: This lab concentrates on medical signal segmentation and classification. Medical signal segmentation is useful for a more accurate analysis of anatomical data by separating the desired areas from the undesirable areas of interest. To segment an image, publicly accessible MRI datasets are used which include edge-based segmentation and unsupervised classification-based segmentation. Classification is used in medical image analysis, which is used to classify images like CT scans, X-rays, MRI scans. Pima Indians Diabetes Database is used to perform medical signal classification using SVMs.

Task 1: MRI Dataset.

The axial view of slice 16, the sagittal view of slice 64 and the coronal view of slice 64 are shown using the 'imshow()' function. The aspect ratio is set to 0.5. In the code below, the data "D" is stored in a NumPy array. In medical imaging, the data obtained is in 3D, which represents the x, y and z axis. To view the Axial slice, which is from top to bottom, the Z - axis is used. In Sagittal view, the slices to be viewed are from left to right or right to left, which correlates with the X-axis. In Coronal view, the slice is to be viewed from front to back, the Y – axis is used. To get a clear picture of the image, "cmap=gray" is used, which lets us view the image in grayscale. To display the plot, "plt.show()" is used.

```
D=io.imread("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/attention
-mri.tif")
#setting the aspect ratio as 0.5
Aspect_ratio = 0.5
#Axial View
      axial_slice = 16
      axial_data = D[:, :, axial_slice]
      plt.imshow(axial_data, cmap='gray' ,aspect=aspect_ratio)
      plt.title(f"Axial View - Slice {axial_slice}")
      plt.show()
#Saggital View
      sagittal_slice = 64
      sagittal_data = D[sagittal_slice, :, :]
      plt.imshow(sagittal_data,cmap='gray', aspect=aspect_ratio)
      plt.title(f"Sagittal View - Slice {sagittal_slice}")
      plt.show()
#Coronal View
      coronal_slice = 64
      coronal_data = D[:, coronal_slice, :].T
      plt.imshow(coronal_data, cmap='gray', aspect=aspect_ratio)
      plt.title(f"Coronal View - Slice {coronal_slice}")
      plt.show()
```

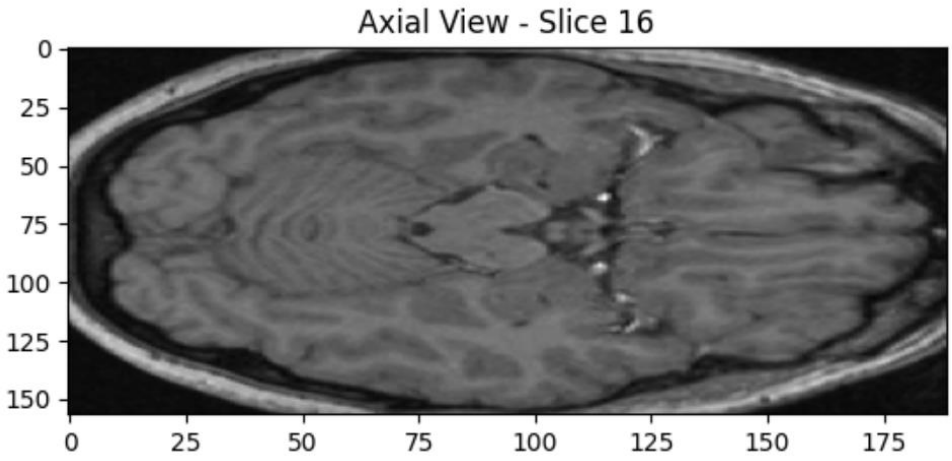

Fig 1: Axial view slice 16

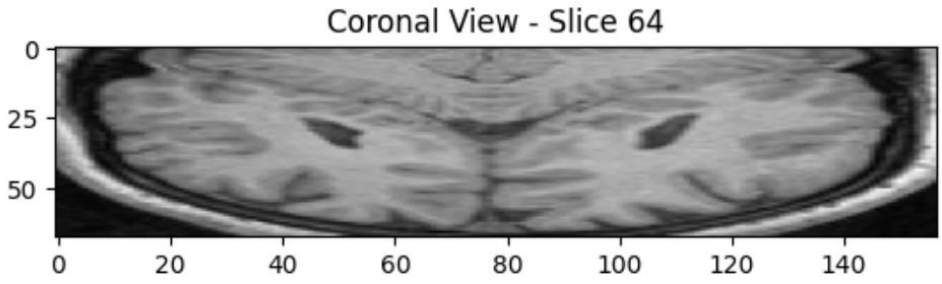

Fig 2: Coronal View slice 64

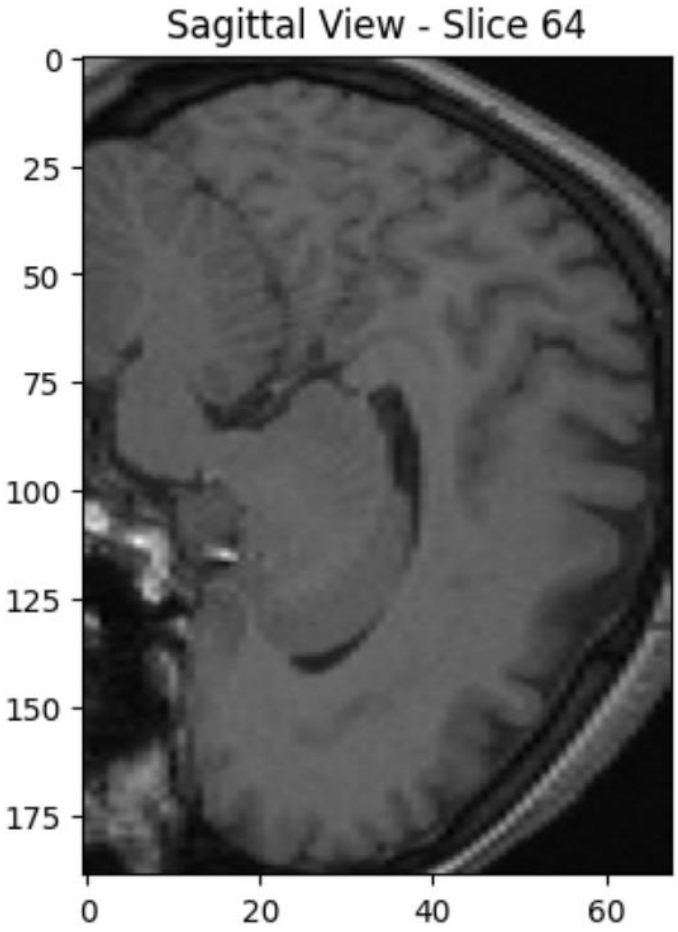

Fig 3: Sagittal View slice 64

Task 2: Edge Filters. 1.

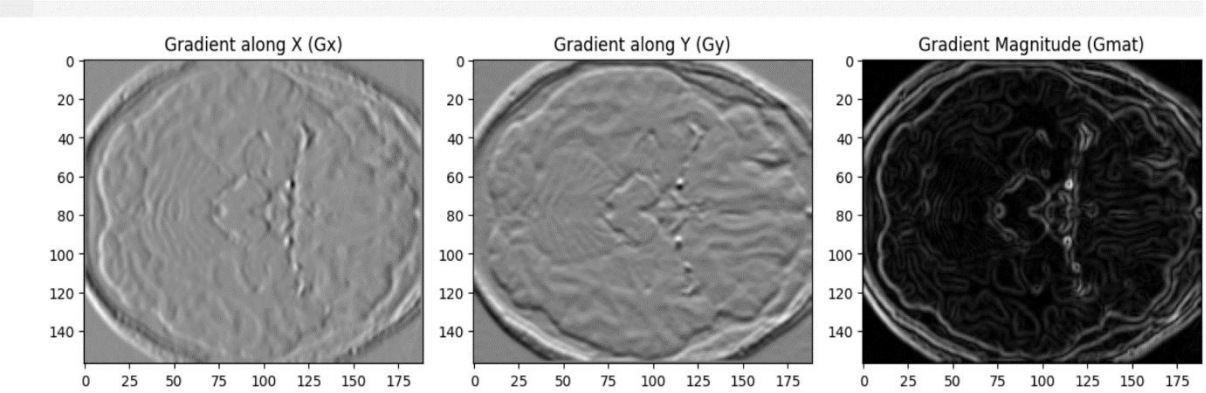

Fig 4: Using edge filters

Filtering is a technique which is used to enhance the features of an image and remove unwanted features. Filtering operations include blurring and color correction.

Code:

```
axial_data = D[:, :, axial_slice]
Gx = cv2.Sobel(axial_data, cv2.CV_64F, 1, 0, ksize=5)
Gy = cv2.Sobel(axial_data, cv2.CV_64F, 0, 1, ksize=5)
Gmat = np.sqrt(Gx**2 + Gy**2)
plt.figure(figsize=(12, 4))
plt.subplot(131)
plt.imshow(Gx, cmap='gray')
plt.title("Gradient along X (Gx)")

plt.subplot(132)
plt.imshow(Gy, cmap='gray')
plt.title("Gradient along Y (Gy)")

plt.subplot(133)
plt.imshow(Gmat, cmap='gray')
plt.title("Gradient Magnitude (Gmat)")

plt.tight_layout()
plt.show()
```

Explanation: The above code is to calculate the image gradients along x and y directions of the Axial view – Slice 16, with the function cv2.Sobel(), and the gradient magnitude using Numpy.sqrt() function, and plot image gradients ("Gx", "Gy") as well as gradient magnitude ("Gmat") using imshow() function. The given dataset “D” is used to calculate the gradient along the two directions. The square root function is used to calculate the gradient along the x and y directions. According to the Pythagorean theorem, the ‘Gmat’ is taken as the hypotenuse and x,y is taken as the sides of a triangle to measure the gradient. ‘Cmap=gray’ is used to show the images in a grayscale. The last step is to show the image using ‘plt.show()’. This is used to show the edges in the provided image (Axial View – Slice 16).

2. Using Prewitt:

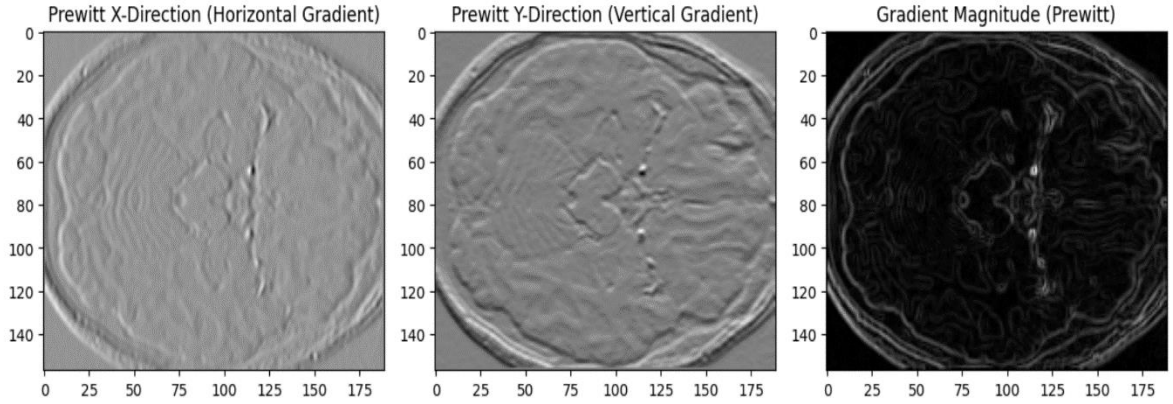

Fig 5: Using Prewitt and detecting edges

Code:

```
Kx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype=np.float32)
Ky = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]], dtype=np.float32)
Gx = cv2.filter2D(axial_data, -1, Kx)
Gy = cv2.filter2D(axial_data, -1, Ky)
Gmag = np.sqrt(Gx**2 + Gy**2)
plt.figure(figsize=(12, 4))
plt.subplot(131)
plt.imshow(Gx, cmap='gray')
plt.title("Prewitt X-Direction (Horizontal Gradient)")

plt.subplot(132)
plt.imshow(Gy, cmap='gray')
plt.title("Prewitt Y-Direction (Vertical Gradient)")

plt.subplot(133)
plt.imshow(Gmag, cmap='gray')
plt.title("Gradient Magnitude (Prewitt)")

plt.tight_layout()
plt.show()
```

The Kx and Ky represent the Prewitt kernels in the X and Y directions. The '-1' is used to specify that the depth in both images must be the same. The 'cmap' is used to obtain the images in gray scale and the figure is plotted using the 'subplot' function and displayed using 'plt.show()'.

3. Conducting canny edge detection:

Canny edge detector is a tool which is used to detect a vast range of edges in an image. Double thresholding is a step in canny edge detection which distinguishes the strong edges and the weak edges and removes noise or distortions in the image. The given upper and lower threshold limits are (2,5) and (3,15). The need for using double thresholding is pixels with values above the upper threshold are taken as strong edges and values between the two thresholds are considered

weak edges. Pixels with values below the lower threshold are not edges and hence are not considered. In the given limits; (2,5), anything below 2 is not considered as an edge and any value above 5 is taken as a strong edge. Values between 2 & 5 are weak edges. The same applies to the other set of points (3,15). The edge detection becomes even more prominent in identifying the edges which were not discoverable before. The sigma function in the code below ensures the image to be smooth.

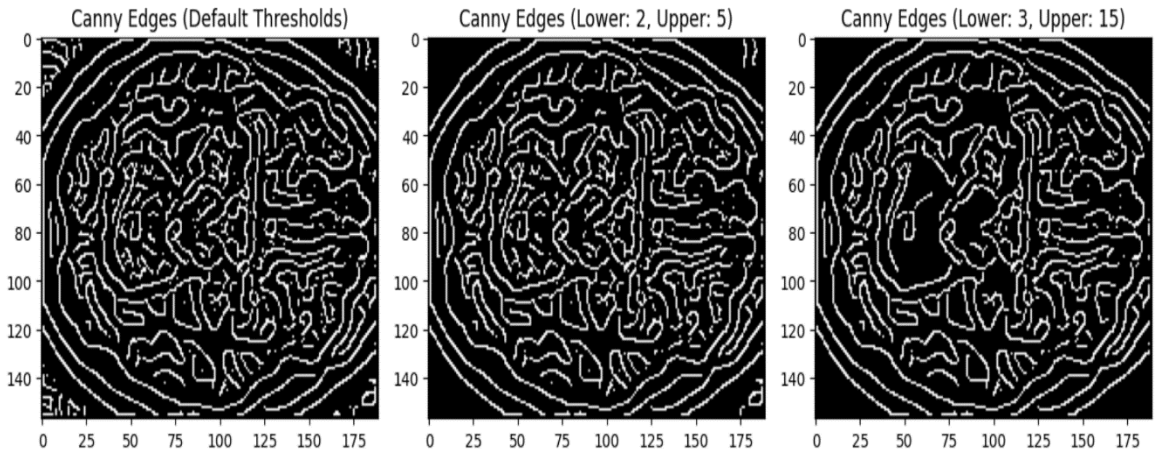

Fig 6: Images when Canny edge detection is used

Code:

```
      edges_default = feature.canny(axial_data, sigma=2)
      edges_lower = feature.canny(axial_data, low_threshold=2,
high_threshold=5, sigma=2)

      edges_upper = feature.canny(axial_data, low_threshold=3,
high_threshold=15, sigma=2)

      # Plot the results
      plt.figure(figsize=(12, 4))
      plt.subplot(131)
      plt.imshow(edges_default, cmap='gray')
      plt.title("Canny Edges (Default Thresholds)")

      plt.subplot(132)
      plt.imshow(edges_lower, cmap='gray')
      plt.title("Canny Edges (Lower: 2, Upper: 5)")

      plt.subplot(133)
      plt.imshow(edges_upper, cmap='gray')
      plt.title("Canny Edges (Lower: 3, Upper: 15)")

      plt.tight_layout()
      plt.show()
```

4. Spatial frequency:

Spatial frequency is referred to as how frequently a sinusoidal component repeats per unit of distance. In image processing, it is a measure of how the pixel value changes when moved across an image. Low-pass filters are used for areas having slow changes in their pixel values, whereas High-pass filters are used for areas having swift changes in the pixel values and are used to detect fine edges and features. Edge filters are high-pass filters as they are required to detect fine features and textures of images which have quick changes in the pixel values. Whereas the low-pass filter is used to smooth, blur images.

Task 3: Kmeans Clustering:

1. Kmeans clustering is a way to separate a data set into K unique clusters. To start the process, the desired number of clusters must be specified, then the algorithm will assign each observation to one of the k-clusters. The distance between data points is calculated and a centroid is given to specific clusters. The difference between supervised and unsupervised learning is that in supervised learning, the input data is labeled, and data is known, it is less complex when compared to unsupervised learning. It gives more accurate results but cannot compute any hidden features on its own. Whereas in unsupervised learning, the input data is not labeled or known, and it is more complex in nature. The accuracy of the results is not high, but it can detect hidden features on its own.

Code: Grouping the pixels in slice 16 using KMeans

```
from sklearn.cluster import KMeans
axial_slice = D[:, :, 16]

pixels = axial_slice.reshape(-1, 1)

n_clusters = [4, 8, 20]

# Performing KMeans to cluster
for n in n_clusters:
    kmeans = KMeans(n_clusters=n, random_state=0)
    labels = kmeans.fit_predict(pixels)

    cluster_centers = kmeans.cluster_centers_

    # Replace pixel values with cluster center values
    clustered_image =cluster_centers[labels].reshape(axial_slice.shape)

    plt.figure()
    plt.imshow(clustered_image, cmap='viridis', aspect=0.5)
    plt.show()
```

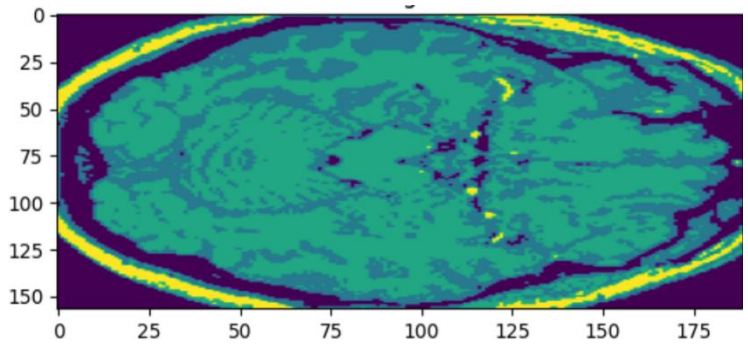

Fig 8: Clustering with 4 clusters

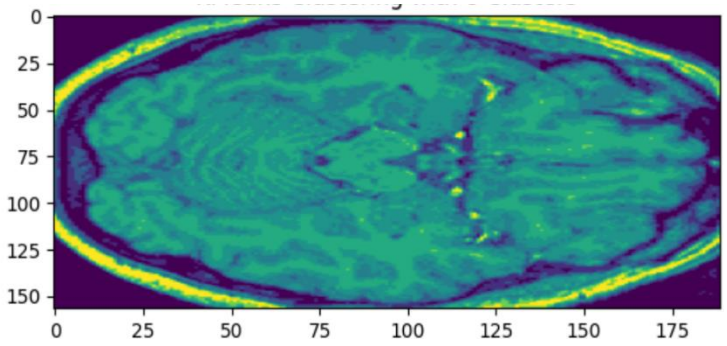

Fig 9: Clustering with 8 clusters

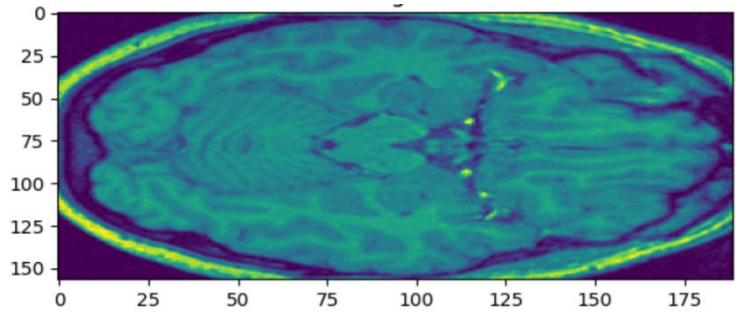

Fig 10: Clustering with 20 clusters

In figure 8, the KMeans function will group the values into 4 different clusters.

In figure 9, the KMeans function will group the values into 8 unique clusters with finer features detected.

In figure 10, the algorithm will group the different values in 20 clusters and the subtle features which weren't detected in 4 and 8 clusters, are visible in the image with 20 clusters.

4. The results vary when the algorithm is repeated several times as a new value for the centroid gets assigned each time the iteration is taken place and hence, new values are obtained.

5. Code: relationship between within-cluster sums of point-to-centroid distances (kmeans.inertia_) and number of iterations (kmeans.n_iter_).

```
      axial_slice = D[:, :, 16]
      #changing into a 1-D vector
      pixels = axial_slice.reshape(-1, 1)

      # Number of clusters
      n_clusters = 4
      inertia_values = []
      n_iterations = []

      # Fit K-Means for a range of iterations
      for i in range(1, 50):
          kmeans = KMeans(n_clusters=n_clusters, random_state=0, max_iter=i)
          kmeans.fit(pixels)
          inertia = kmeans.inertia_
          n_iter = kmeans.n_iter_
          inertia_values.append(inertia)
          n_iterations.append(n_iter)

      # Plot the relationship
      plt.figure()
      plt.plot(n_iterations, inertia_values, marker='o', linestyle='-')
      plt.xlabel('Number of Iterations')
      plt.ylabel('Within-Cluster Sums of Distances (Inertia)')
      plt.title(f'K-Means Inertia vs. Number of Iterations
(Clusters={n_clusters})')
      plt.grid(True)
      plt.show()
```

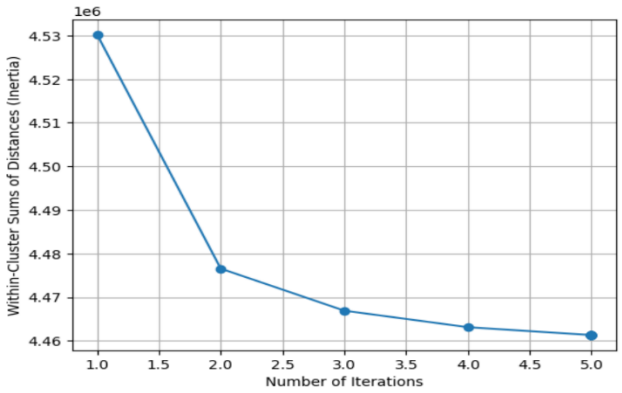

Fig 11: relationship between within-cluster sums of point-to-centroid distances (kmeans.inertia_) and number of iterations (kmeans.n_iter_).

Task 4: Support Vector Machines:

The SVM (Support Vector Machine) is a supervised learning model. It is used to separate data into two different classes using a hyperplane, mostly used for classification and regression problems. This model can be used for linear and non-linear data. The hyperplane is also known as the decision boundary.

Code:

```
import os
from csv import reader
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import svm

# Load data
with open("pima.csv") as f:
    csv_data = reader(f, delimiter=',')
    raw_data = np.array(list(csv_data),dtype=object)

# Preprocess data
data_x = []
data_y = []
tuple_len = len(raw_data[0])
for i in raw_data:
    if not i:
        continue
```

```
            data_x.append([float(j) for j in i[0:tuple_len - 2]])
            if i[tuple_len - 1] == "yes":
                data_y.append(1)
            else:
                data_y.append(0)
        # Split dataset
        x_train, x_test, y_train, y_test = train_test_split(
            data_x, data_y, test_size=0.33, random_state=73)
        #TODO:
        from sklearn.metrics import accuracy_score
        svm_classifier = svm.SVC(kernel='linear', random_state=0)

        svm_classifier.fit(x_train, y_train)

        y_pred = svm_classifier.predict(x_test)

        accuracy = accuracy_score(y_test, y_pred)
        print(f'Accuracy: {accuracy:.2f}')
```

```
        OUTPUT: Accuracy: 0.74
```

An accuracy of 0.74 is obtained, which means that the SVM classified 74% of the data correctly. In some cases, this accuracy would be sufficient, while in others, more accuracy is required. If one class is more than the other, then predicting the majority class can give higher accuracy.

Conclusion:

In the first task of the lab, the axial view of slice 16 and sagittal and coronal views of slice 64 is viewed. In the next part, the image gradients are viewed using the Sobel function and Prewitt edge filtering is performed. Canny edge detection is implemented using lower and upper thresholds. In the third part, the pixels in slice 16 are clustered in three different clusters. In the last part of the lab, machine learning techniques like SVM has been used to predict Diabetes in patients.