Credit Hours System

HEMN451 – Medical Pattern Recognition

Cairo University

Faculty of Engineering

Midterm Report

Team Members:

- Hoda Hossam [1170156]
- Mohammed Nader [1170366]
- Tasneem Adel [1162182]

Contents

What is Image Segmentation:

Image segmentation is the process of partitioning of digital images into various parts or regions (of pixels) reducing the complexities of understanding the images to machines. Image segmentation could also involve separating the foreground from the background or assembling of pixels based on various similarities in the color or shape.

Various image segmentation algorithms are used to split and group a certain set of pixels together from the image. It is actually the task of assigning the labels to pixels and the pixels with the same label fall under a category where they have some or the other thing common in them.

And using these labels, you can specify boundaries, draw lines, and separate the most required objects in an image from the rest of the unimportant one.

In machine learning image segmentation helps to make these identified labels further use for supervised and unsupervised training that is mainly required to develop machine learning based AI model. Image segmentation is used for image processing into various types of computer vision projects.

In image recognition system, segmentation is an important stage that helps to extract the object of interest from an image which is further used for processing like recognition and description. Image segmentation is the practice for classifying the image pixels.

Problem-1 (Fuzzy-C-means):

Algorithm:

This algorithm works by assigning membership to each data point corresponding to each cluster center on the basis of distance between the cluster center and the data point. More the data is near to the cluster center more is its membership towards the particular cluster center. Clearly, summation of membership of each data point should be equal to one.

After each iteration membership and cluster centers are updated according to the formula:

$$\mu_{ij} = 1 / \sum_{k=1}^{c} (d_{ij} / d_{ik})^{(2/m-1)}$$

$$v_j = (\sum_{i=1}^{n} (\mu_{ij})^m x_i) / (\sum_{i=1}^{n} (\mu_{ij})^m), \forall j = 1, 2, ...c$$

where,

- *'n'* is the number of data points.
- *'m'* is the fuzziness index m € [1, ∞].
- *'μij'* represents the membership of i^{th} data to j^{th} cluster center.
- *'vj'* represents the j^{th} cluster center.
- *'c'* represents the number of cluster center.
- *'dij'* represents the Euclidean distance between i^{th} data and j^{th} cluster center.

Main objective of fuzzy c-means algorithm is to minimize:

$$J(U,V) = \sum_{i=1}^{n} \sum_{j=1}^{c} (\mu_{ij})^m \left\| x_i - v_j \right\|^2$$

where,

- *'||xi – vj||'* is the Euclidean distance between *ith* data and *jth* cluster center.

Algorithmic steps for Fuzzy c-means clustering

Let X = {x_1, x_2, x_3 ..., x_n} be the set of data points and V = {v_1, v_2, v_3 ..., v_c} be the set of centers.

1) Randomly select *'c'* cluster centers.

2) Calculate the fuzzy membership *'μ_{ij}'* using:

$$\mu_{ij} = 1 / \sum_{k=1}^{c} (d_{ij} / d_{ik})^{(2/m-1)}$$

3) Compute the fuzzy centers *'vj'* using:

$$v_j = (\sum_{i=1}^{n} (\mu_{ij})^m x_i) / (\sum_{i=1}^{n} (\mu_{ij})^m), \forall j = 1, 2, \ldots . c$$

4) Repeat step 2) and 3) until the minimum *'J'* value is achieved or $||U(k+1) - U(k)|| < \beta$.

where,

- 'U = (μij)n*c' *is the fuzzy membership matrix.*
- 'J' *is the objective function.*
- 'k' *is the iteration step.*
- 'β' *is the termination criterion between [0, 1].*

Code:

First initialize important variables:

```
self.maximum_iteration = 500
self.epsilon = 0.05
self.m = 2   # fuzziness index which is usually set to 2
```

In our algorithm we work on the Gray scale image then after the algorithm outputs the images we loop over it and change every pixel associated to a cluster with the mean value of pixels in that cluster, we choose epsilon = 0.05 as it gave us a better accuracy in the resulted image and a faster response, maximum iteration of the algorithm is 500 iterations after that it will output the final result it got.

```
self.membership_degree = np.zeros((self.pixelNum, self.clusterNumber))
idx = np.arange(self.pixelNum)
for cluster in range(self.clusterNumber):
    idxii = idx%self.clusterNumber==cluster
    self.membership_degree[idxii,cluster] = 1

self.Cluster_center = np.linspace(np.min(self.img_array_Cmeans_BW),np.max(self.img_array_Cmeans_BW),self.clusterNumber)
self.Cluster_center = self.Cluster_center.reshape(self.clusterNumber,1)
```

We initialize membership degree to every pixel in the photo to every cluster that the user choose, and initialize the centres with number of clusters identified by the user.

```
def update_membershipDegree(self):
    c_mesh,x_mesh = np.meshgrid(self.Cluster_center,self.flatten)
    power = 2./(self.m-1)
    p1 = abs(x_mesh-c_mesh)**power  # Ecliden distance between cluster center and data
    p2 = np.sum((1./p1),axis=1)
    return 1./(p1*p2[:,None])

def update_Centers(self):
    num = np.dot(self.flatten,self.membership_degree**self.m)
    den = np.sum(self.membership_degree**self.m,axis=0)
    return num/den
```

Function to update the cluster centers and the membership degree of each pixle according to the equations stated above.

```
def c_mean_operation(self):
    i = 0
    while True:
        self.Cluster_center = self.update_Centers()
        old_degree = np.copy(self.membership_degree)
        self.membership_degree = self.update_membershipDegree()
        difference = np.sum(abs(self.membership_degree - old_degree))
        printed = str(i) + " - difference = " + str(format(difference, '.4f'))
        self.ui.IterationsLable.setText(printed)

        if difference < self.epsilon or i > self.maximum_iteration:
            break
        i += 1
    self.Segmented_Image = np.argmax(self.membership_degree, axis=1)
    self.Segmented_Image = self.GetSegmentedImage_Cmeans(self.Segmented_Image,cv2.cvtColor(self.img_array_Cmeans_original, cv2.COLOR_BGR2RGB))
```

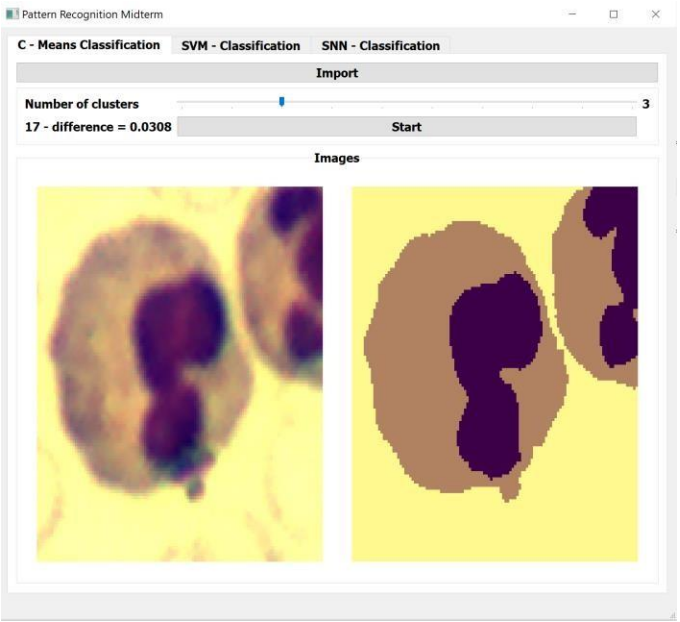
The loop starts updating cluster centers and saves the value of the old membership degree of the pixel, then updates the membership degree and compare it to the old degree which is the difference that our algorithm will stop if it's less than epsilon value (0.05).

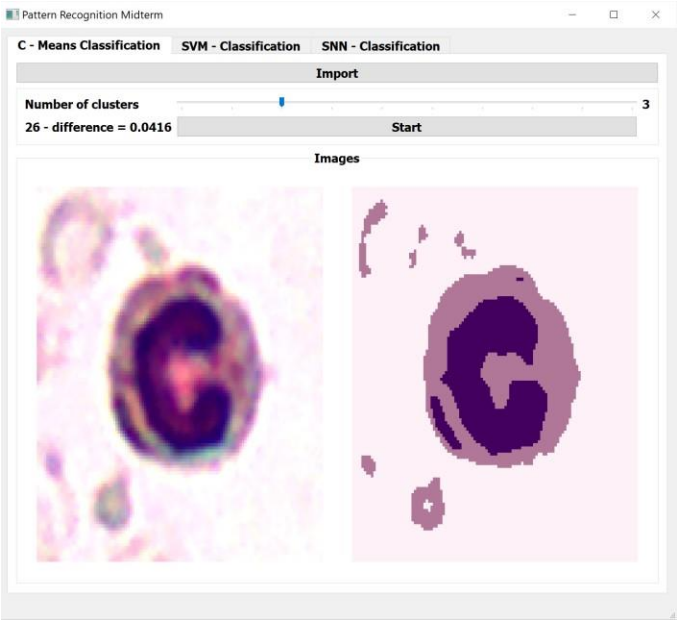
The output of the algorithm is an array containing the predection of the cluster regarding each pixel Ex: y = [1,1,2,3,0], remember that we are working on the grey scale image, then to get the mean of pixels we get all the pixels of class 1 and get the mean of them and finally change the value 1 in array y to be this mean, and plot the image.

```
def GetSegmentedPhoto(self,y,x):
    cluster_mean  = []
    cluster_mean2 = []
    cluster_mean3 = []

    for i in range(len(x)):
        if y[i] == 1:
            cluster_mean.append(x[i])
        if y[i] == 2:
            cluster_mean2.append(x[i])
        if y[i] == 3:
            cluster_mean3.append(x[i])

    cluster_mean = np.mean(cluster_mean,axis=0)
    cluster_mean2 = np.mean(cluster_mean2,axis=0)
    cluster_mean3 = np.mean(cluster_mean3,axis=0)
    y.reshape(-1,1)
    y_mean = []
    for i in y:
        if i == 1:
            y_mean.append(cluster_mean)
        if i == 2:
            y_mean.append(cluster_mean2)
        if i == 3:
            y_mean.append(cluster_mean3)

    y_mean = np.array(y_mean)
    reshape_vale=int(math.sqrt(y.shape[0]))

    return(y_mean.reshape(reshape_vale,reshape_vale,3))
```

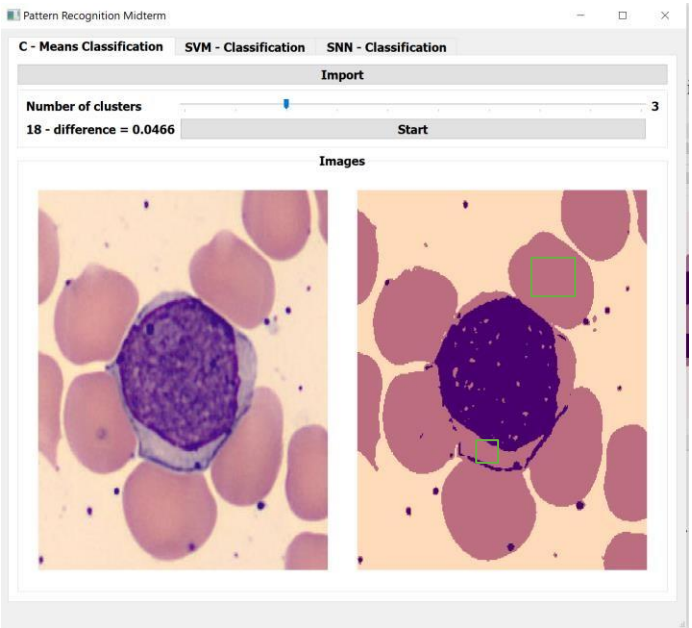
Outputs and discussion:

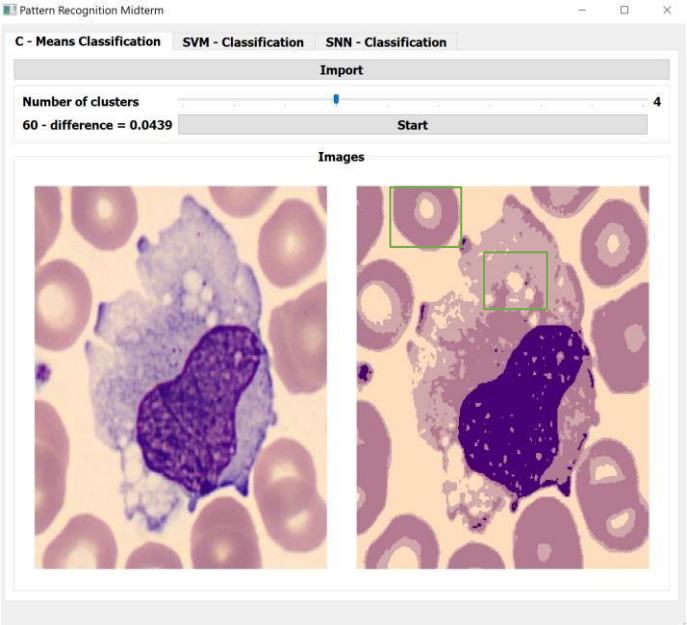
We think that the output of this algorithm is the best one of the algorithm we implemented as it operates very fast on the image and it's output is very consistent.





Yet in some images it cannot differentiate between the outer cell of the white blood cell and the red blood cells but as we will see it's a problem in all algorithms.



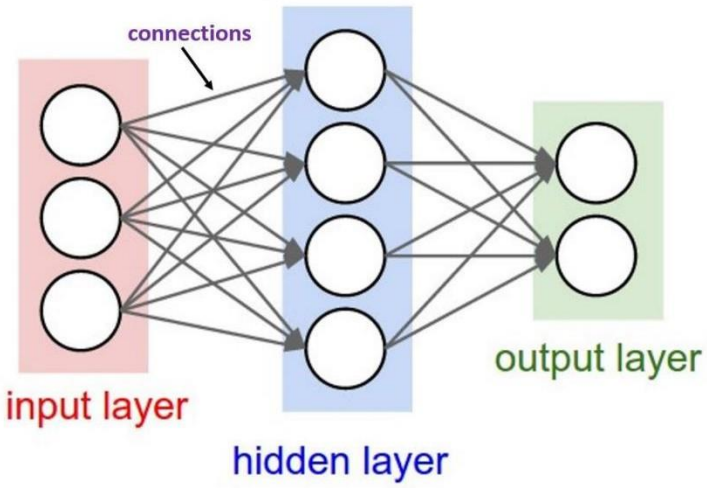


Problem-2 (SNN):

Algorithm:

In this work we implement a Simple Neural Network (SNN), with one hidden layer, that performs multi-label classification. The error function used in our multi-label classification neural network is the Cross-entropy, and is given by the following equation:

$$E(\mathbf{w}) = \sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} log y_{nk} - \frac{\lambda}{2}||\mathbf{w}||^2$$



where y_{nk} is the activation function of the output neurons, in our case Softmax, and is given by:

$$y_{nk} = \frac{e^{(\mathbf{w}_k^{(2)})^T \mathbf{z}_n}}{\sum_{j=1}^{K} e^{(\mathbf{w}_j^{(2)})^T \mathbf{z}_n}}$$

The hidden layer neurons use sigmoid is given by:

$$f(t) = \frac{1}{1+e^{-t}}$$

Note that:

- The subscript k denotes the output layer.
- The subscript j denotes the hidden layer.
- The subscript i denotes the input layer.
- w_{kj} denotes a weight from the hidden to the output layer.
- w_{ji} denotes a weight from the input to the hidden layer.
- w_{j0}, w_{k0} denote the bias values.
- t denotes a target value.
- x denotes an input value.
- h denotes an activation function.
- out denotes an activation value.
- net denotes the net input (weighted sum).

The Forward Pass is given by the following equation:

$$y_k(\mathbf{x},\mathbf{w}) = h_k\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

where h_k represents the outer activation function, which is Softmax in our case, and h represents the hidden layer's activation function which is sigmoid.

The Backwards Pass is used to update input and output weights that are stored respectively in the arrays W(1) & W(2) in the network, so that they cause the actual output to be closer with the target output. Thereby, minimizing the error for each output neuron and the network as a whole. The gradient ascent algorithm is used to estimate the proper step values(deltas) to update both W(1) , W(2). The algorithm steps are:

Algorithm

1. Initialization: $k = 1$, $\mathbf{w}^{(k)}$, $\eta > 0$.
2. Parameter update

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta \nabla_{\mathcal{L}(\mathbf{w}^{(k)})}$$

3. Set $k = k + 1$ and go to step 2 or terminate (η is called learning rate)

Code:

First initialize weights and layers of the NN, the learning rate is 0.07 as it gets the best accuracy, number of neurons is 128.

```
neurons = 128
self.lr = 0.07
ip_dim = x.shape[1]
op_dim = y.shape[1]

self.w1 = np.random.randn(ip_dim, neurons)
self.b1 = np.zeros((1, neurons))
self.w2 = np.random.randn(neurons, op_dim)
self.b2 = np.zeros((1, op_dim))
```

define sigmoid, softmax, cross entropy error functions to be used in back and forward propagation .

```
def sigmoid(s):
    return 1/(1 + np.exp(-s))

def sigmoid_derv(s):
    return s * (1 - s)

def softmax(s):
    exps = np.exp(s - np.max(s, axis=1, keepdims=True))
    return exps/np.sum(exps, axis=1, keepdims=True)

def cross_entropy(pred, real):
    n_samples = real.shape[0]
    res = pred - real
    return res/n_samples

def error(pred, real):
    n_samples = real.shape[0]
    logp = - np.log(pred[np.arange(n_samples), real.argmax(axis=1)])
    loss = np.sum(logp)/n_samples
    return loss
```

implemented the forward and back propagation function as the stated above:

```
def feedforward(self):
    z1 = np.dot(self.x, self.w1) + self.b1
    self.a1 = sigmoid(z1)
    z2 = np.dot(self.a1, self.w2) + self.b2
    self.a2 = softmax(z2)

def backprop(self):
    loss = error(self.a2, self.y)
    # print('Error :', loss)
    a2_delta = cross_entropy(self.a2, self.y) # w2
    z1_delta = np.dot(a2_delta, self.w2.T)
    a1_delta = z1_delta * sigmoid_derv(self.a1) # w1

    self.w2 -= self.lr * np.dot(self.a1.T, a2_delta)
    self.b2 -= self.lr * np.sum(a2_delta, axis=0, keepdims=True)
    self.w1 -= self.lr * np.dot(self.x.T, a1_delta)
    self.b1 -= self.lr * np.sum(a1_delta, axis=0)
```

we created a new dataset from the given images containing a samples from each object in the image the inner cell and the outer cell and the background'cytoplasm' to give it to the algorithm as X with Y as it's classes, this will make the model train on detecting and classifying these objects, then in predict function it will iterate

over each pixel of the image given X_Test to classify each pixel to one of these objects.

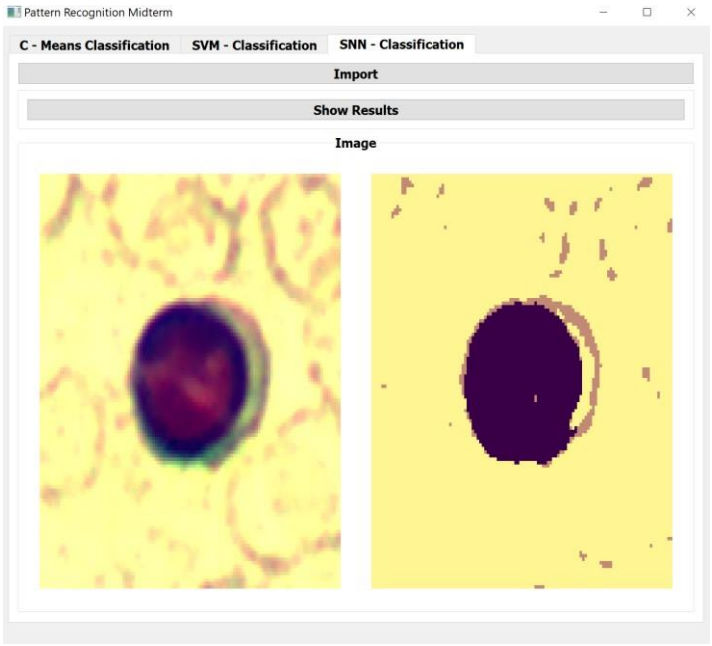
```
def predict(self, data):
    predected = []
    for i in data:
        self.x = i
        self.feedforward()
        predected.append(self.a2.argmax())
    return predected
```

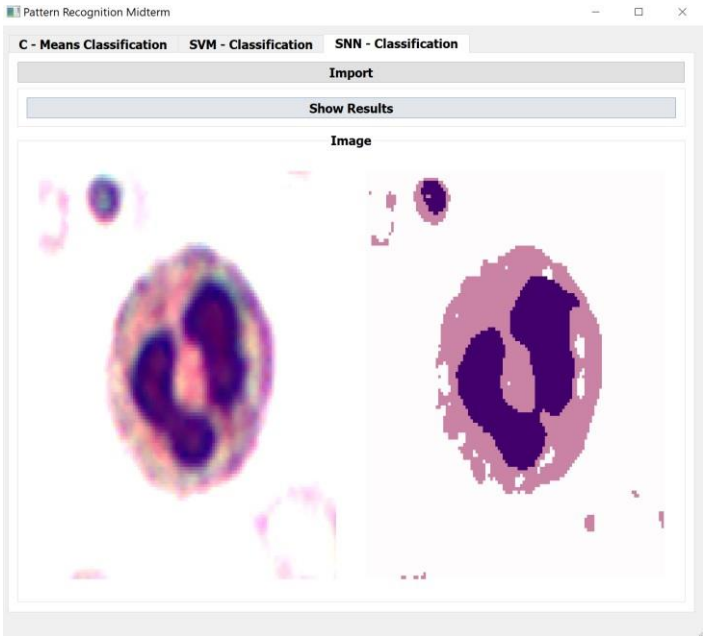
Then the output predicted array just as the C-means algorithm we pass it to a function to colour it using this function:

```
def GetSegmentedPhoto(self,y,x):
    cluster_mean  = []
    cluster_mean2 = []
    cluster_mean3 = []

    for i in range(len(x)):
        if y[i] == 1:
            cluster_mean.append(x[i])
        if y[i] == 2:
            cluster_mean2.append(x[i])
        if y[i] == 3:
            cluster_mean3.append(x[i])

    cluster_mean = np.mean(cluster_mean,axis=0)
    cluster_mean2 = np.mean(cluster_mean2,axis=0)
    cluster_mean3 = np.mean(cluster_mean3,axis=0)
    y.reshape(-1,1)
    y_mean = []
    for i in y:
        if i == 1:
            y_mean.append(cluster_mean)
        if i == 2:
            y_mean.append(cluster_mean2)
        if i == 3:
            y_mean.append(cluster_mean3)

    y_mean = np.array(y_mean)
    reshape_vale=int(math.sqrt(y.shape[0]))

    return(y_mean.reshape(reshape_vale,reshape_vale,3))
```

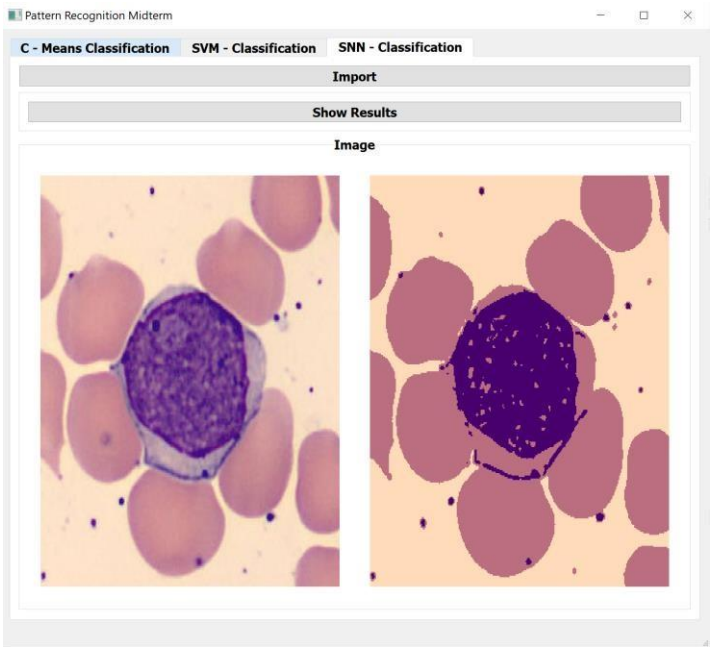
Outputs and discussion:

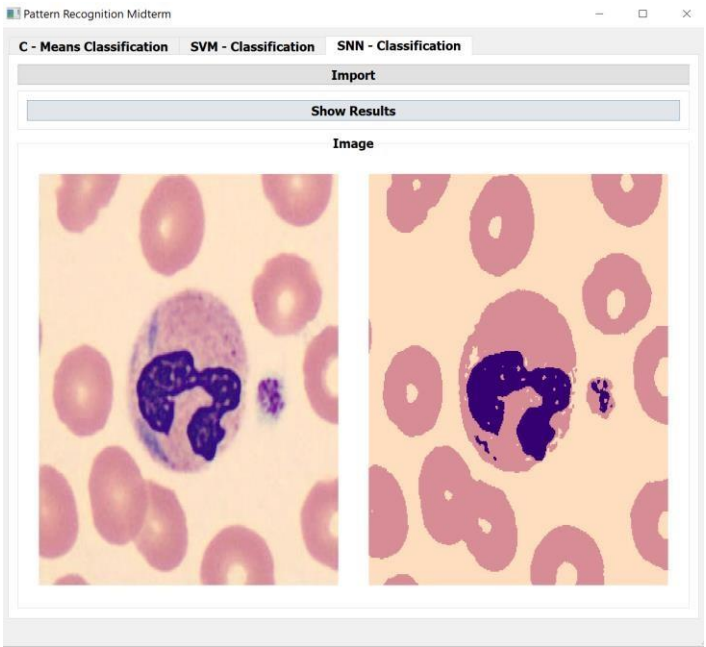
The output of the algorithm is not as clear as the c-means algorithm due to our choice of the train set as we choose it randomly, but still, its accuracy is more than 90% in most of the pictures.





Yet just like c-means it hard for it to distinguish between the inner cell of WBC and the Red blood cells.

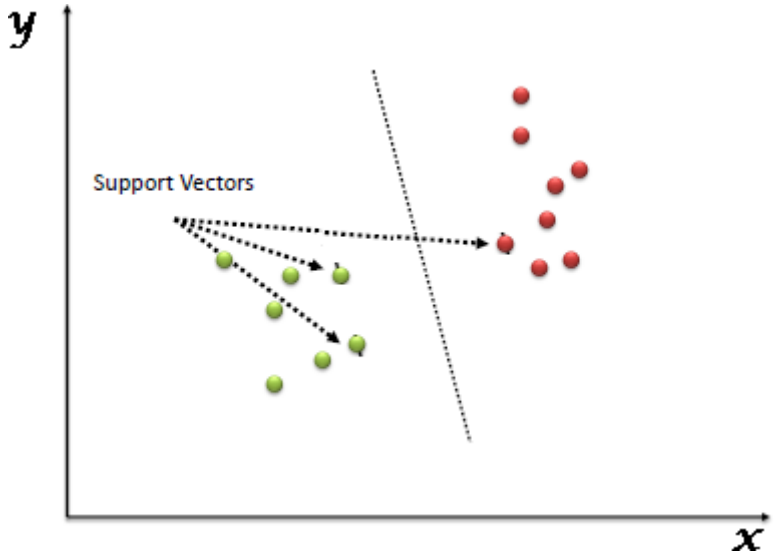




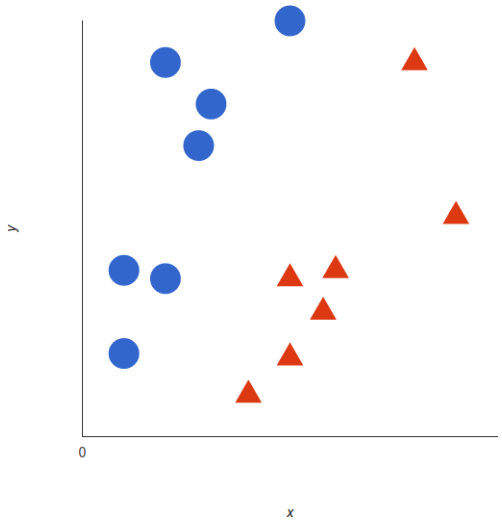
Problem-3 (SVM):

Algorithm:

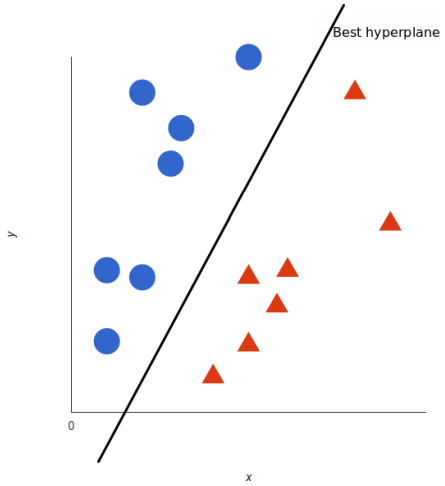
"Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is number of features 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 very well.



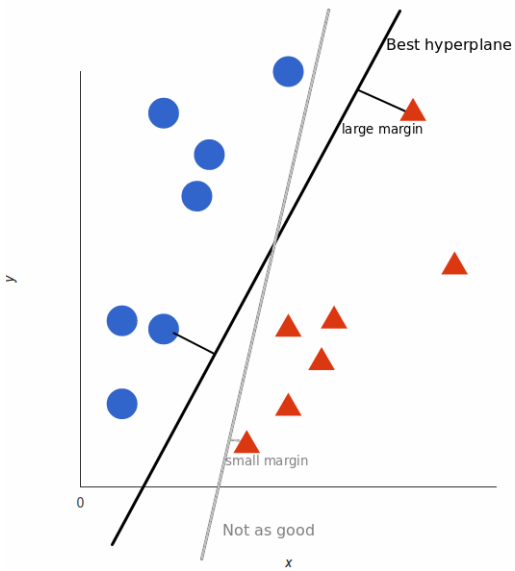
The basics of Support Vector Machines and how it works are best understood with a simple example. Let's imagine we have two tags: red and blue, and our data has two features: x and y. We want a classifier that, given a pair of (x,y) coordinates, outputs if it's either red or blue. We plot our already labeled training data on a plane:



A support vector machine takes these data points and outputs the hyperplane (which in two dimensions it's simply a line) that best separates the tags. This line is the decision boundary: anything that falls to one side of it we will classify as blue, and anything that falls to the other as red.



But, what exactly is the best hyperplane? For SVM, it's the one that maximizes the margins from both tags. In other words: the hyperplane (remember it's a line in this case) whose distance to the nearest element of each tag is the largest.



Code:

First, we imported the SVM model from the sklearn library, and just like the NN we passed the same images we got from the dataset for every object we wanted it to classify (outer_cell, Inner_cell, backgound ‘cytoplasm’) to train the model to them.

Then, we called this code which will tell us the best parameters of the SVM:

```
# parameter = [
#     {'C':[1,10,100,1000],'kernel':['linear']},
# ]
# clf = GridSearchCV(estimator = svm.SVC(), param_grid = parameter ,n_jobs = -1)

# clf.fit(x,y)

# print("best score is ", clf.best_score_)
# print('best "c" is ', clf.best_estimator_.C)
# print('best kernel is ',clf.best_estimator_.kernel)
# print('best gamma is ',clf.best_estimator_.gamma)
```

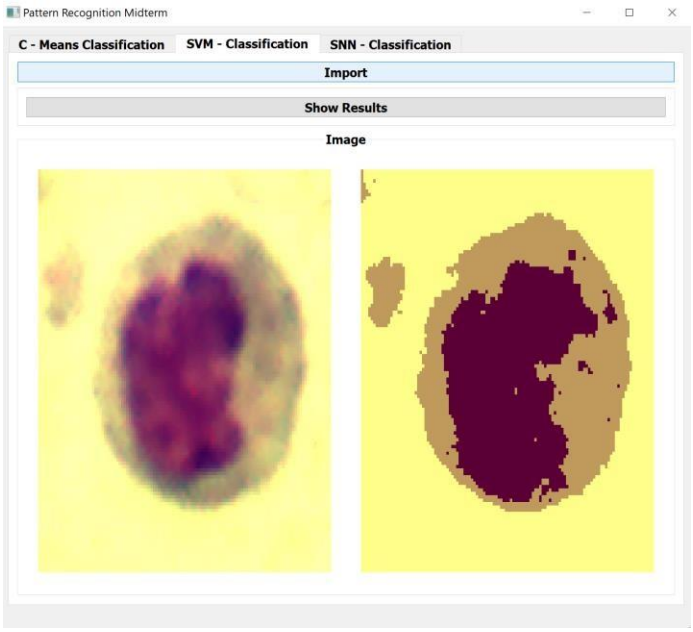
Which was (C=100 , Kernel = ‘linear’, Gamma = ‘scale’), then we fit the model to our data:

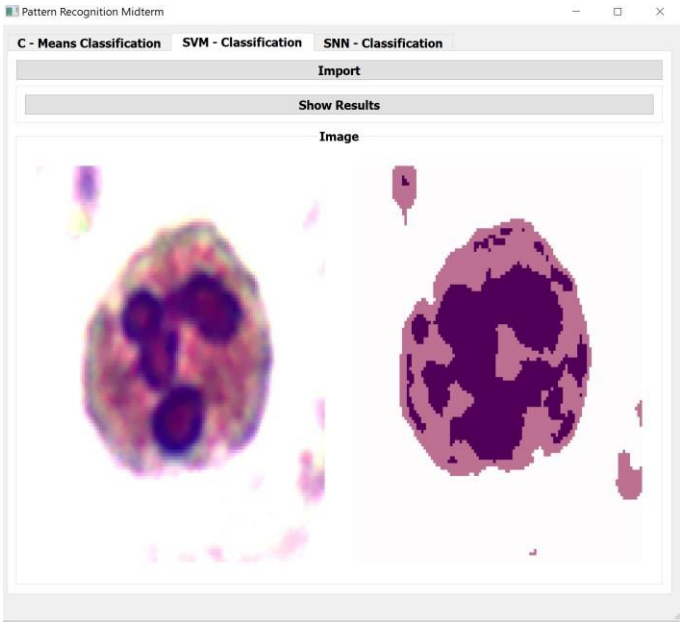
```
clf = svm.SVC(C=100,kernel='linear',gamma="scale")
clf.fit(x,y)
```

Then, pass the image to svm.predict() to get the output array, which will be just like other algorithms, so we call the same function stated above to colour the image with the mean of colours.

```
def GetSegmentedPhoto(self,y,x):
    cluster_mean  = []
    cluster_mean2 = []
    cluster_mean3 = []

    for i in range(len(x)):
        if y[i] == 1:
            cluster_mean.append(x[i])
        if y[i] == 2:
            cluster_mean2.append(x[i])
        if y[i] == 3:
            cluster_mean3.append(x[i])

    cluster_mean = np.mean(cluster_mean,axis=0)
    cluster_mean2 = np.mean(cluster_mean2,axis=0)
    cluster_mean3 = np.mean(cluster_mean3,axis=0)
    y.reshape(-1,1)
    y_mean = []
    for i in y:
        if i == 1:
            y_mean.append(cluster_mean)
        if i == 2:
            y_mean.append(cluster_mean2)
        if i == 3:
            y_mean.append(cluster_mean3)

    y_mean = np.array(y_mean)
    reshape_vale=int(math.sqrt(y.shape[0]))

    return(y_mean.reshape(reshape_vale,reshape_vale,3))
```

Outputs and discussion:





Just like the other algorithms this model predicts that the red blood cells are the same as the outer cell of the white blood cells.