Section:KM101

PROJECT

ON

Design and Implementation of Neural Network (online customers purchasing intention dataset)

Submitted by

Sr.No	NAME	Reg.No
1.	**Manideep Reddy**	**11714932**
2.	**Shubham Kumar**	**11716437**
3.	**Yash Shinde**	**11716990**
4.	**Yogesh Nain**	**11717449**

Program Name: B.Tech(CSE)

Under the Guidance of

Mr. Sanjay Kumar Singh

School of Computer Science & Engineering Lovely Professional University, Phagwara

(June-July, 2019)

Section:KM101

DECLARATION

I hereby declare that the project work entitled **"Design and Implementation of Neural Network (online customers purchasing intention dataset)"** submitted to the Lovely Professional University, is a record of an original work done by us under the guidance of Mr Sanjay Kumar Singh ,and this project work has not performed the basis for the award of any Degree or diploma/ associate ship/fellowship and similar project if any, during August to November, 2019.

Name of Student 1 = Manideep Reddy

Registration No. = 11714932

Roll NO. = RKM101A17 Signature of student

Name of Student 2 = Shubham Kumar

Registration No. = 11716437

Roll NO. =RKM101A18 Signature of student

Name of Student 3 = Yash Shinde

Registration No. = 11716990

Roll NO. = RKM101A19 Signature of student

Name of Student 4 = Yogesh Nain

Registration No. = 11717449

Roll NO. = RKM101A20 Signature of student

Section:KM101

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind of support and help of many individuals. We would like to extend our sincere thanks to all of them.

We would like to express our special thanks of gratitude to our teacher and school who gave us the golden opportunity to do this wonderful project on Perceptron, SVM, LVQ, SOM, which also helped us in doing a lot of Research and we came to know about so many new things we are really thankful to them.

Secondly, we would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

ROLES AND RESPONSIBILITY

We divided the project into four parts-

Manideep Reddy	Shubham Kumar	Yash Shinde	Yogesh Nain
Perceptron	Support Vector Machine	Learning Vector Quantization	Self-organizing Map

Section:KM101

DATA SET

Data set information:

The dataset consists of feature vectors belonging to 12,330 sessions.
The dataset was formed so that each session
would belong to a different user in a 1-year period to avoid
any tendency to a specific campaign, special day, user
profile, or period.

Attribute information:

The dataset consists of 10 numerical and 8 categorical attributes.
The 'Revenue' attribute can be used as the class label.

"Administrative", "Administrative Duration", "Informational", "Informational Duration", "Product Related" and "Product Related Duration" represent the number of different types of pages visited by the visitor in that session and total time spent in each of these page categories. The values of these features are derived from the URL information of the pages visited by the user and updated in real time when a user takes an action, e.g. moving from one page to another. The "Bounce Rate", "Exit Rate" and "Page Value" features represent the metrics measured by "Google Analytics" for each page in the e-commerce site.

The value of "Bounce Rate" feature for a web page refers to the percentage of visitors who enter the site from that page and then leave ("bounce") without triggering any other requests to the analytics server during that session. The value of "Exit Rate" feature for a specific web page is calculated as for all pageviews to the page, the percentage that were the last in the session. The "Page Value" feature represents the average value for a web page that a user visited before completing an e-commerce transaction. The "Special Day" feature indicates the closeness of the site visiting time to a specific special day (e.g. Mother’s Day, Valentine's Day) in which the sessions are more likely to be finalized with transaction. The value of

Section:KM101

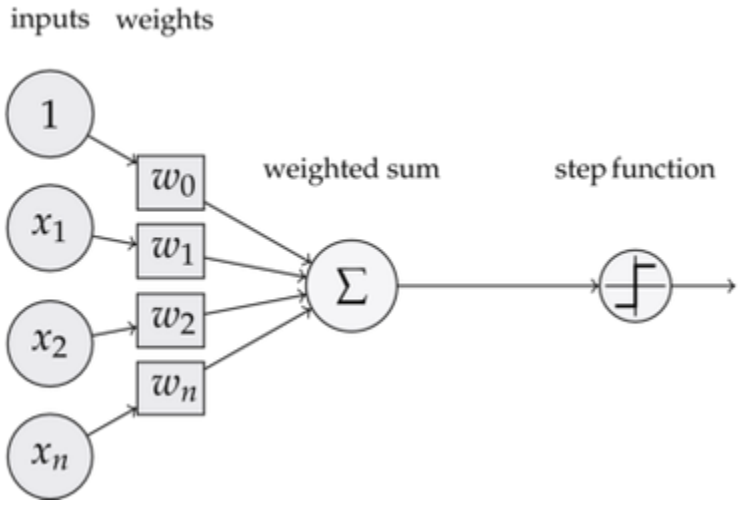
this attribute is determined by considering the dynamics of e-commerce such as the duration between the order date and delivery date. For example, for Valentina's day, this value takes a nonzero value between February 2 and February 12, zero before and after this date unless it is close to another special day, and its maximum value of 1 on February 8. The dataset also includes operating system, browser, region, traffic type, visitor type as returning or new visitor, a Boolean value indicating whether the date of the visit is weekend, and month of the year.

Abstract: Of the 12,330 sessions in the dataset, 84.5% (10,422) were negative class samples that did not end with shopping, and the rest (1908) were positive class samples ending with shopping.

Section:KM101

Perceptron

In machine learning, the **perceptron** is an algorithm for supervised learning of binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class.It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.



Following are the major components of a perceptron:

- **Input**: All the features become the input for a perceptron. We denote the input of a perceptron by [x1, x2, x3, ..,xn], where x represents the feature value and n represents the total number of features. We also have special kind of input called the bias. In the image, we have described the value of the BIAS as w0.
- **Weights**: The values that are computed over the time of training the model. Initially, we start the value of weights with some initial value and these values get updated for each training error. We represent the weights for perceptron by [w1,w2,w3,.. wn].

- **Bias**: A bias neuron allows a classifier to shift the decision boundary left or right. In algebraic terms, the bias neuron allows a classifier to translate its decision boundary. It aims to "move every point a constant distance in a specified direction." Bias helps to train the model faster and with better quality.
- **Weighted summation**: Weighted summation is the sum of the values that we get after the multiplication of each weight [wn] associated with the each feature value [xn]. We represent the weighted summation by $\sum w_i x_i$ for all i -> [1 to n].
- **Step/activation function**: The role of activation functions is to make neural networks nonlinear. For linear classification, for example, it becomes necessary to make the perceptron as linear as possible.
- **Output**: The weighted summation is passed to the step/activation function and whatever value we get after computation is our predicted output.

Training Algorithm

Perceptron network can be trained for single output unit as well as multiple output units.

Training Algorithm for Single Output Unit

Step 1 − Initialize the following to start the training −

- Weights
- Bias
- Learning rate alpha

For easy calculation and simplicity, weights and bias must be set equal to 0 and the learning rate must be set equal to 1.

Step 2 − Continue step 3-8 when the stopping condition is not true.

Step 3 − Continue step 4-6 for every training vector **x**.

Step 4 − Activate each input unit as follows −

x_{i}\:=\:s_{i}\:(i\:=\:1\:to\:n)

Step 5 − Now obtain the net input with the following relation −

y_{in}\:=\:b\:+\:\displaystyle\sum\limits_{i}^n x_{i}.\:w_{i}

Here **‘b’** is bias and **‘n’** is the total number of input neurons.

Section:KM101

Step 6 − Apply the following activation function to obtain the final output.

f(y_{in})\:=\:\begin{cases}1 & if\:y_{in}\:>\:\theta\\0 & if \: -\theta\:\leqslant\:y_{in}\:\leqslant\:\theta\\-1 & if\:y_{in}\:<\:-\theta \end{cases}

Step 7 − Adjust the weight and bias as follows −

Case 1 − if **y ≠ t** then,

w_{i}(new)\:=\:w_{i}(old)\:+\:\alpha\:tx_{i}

b(new)\:=\:b(old)\:+\:\alpha t

Case 2 − if **y = t** then,

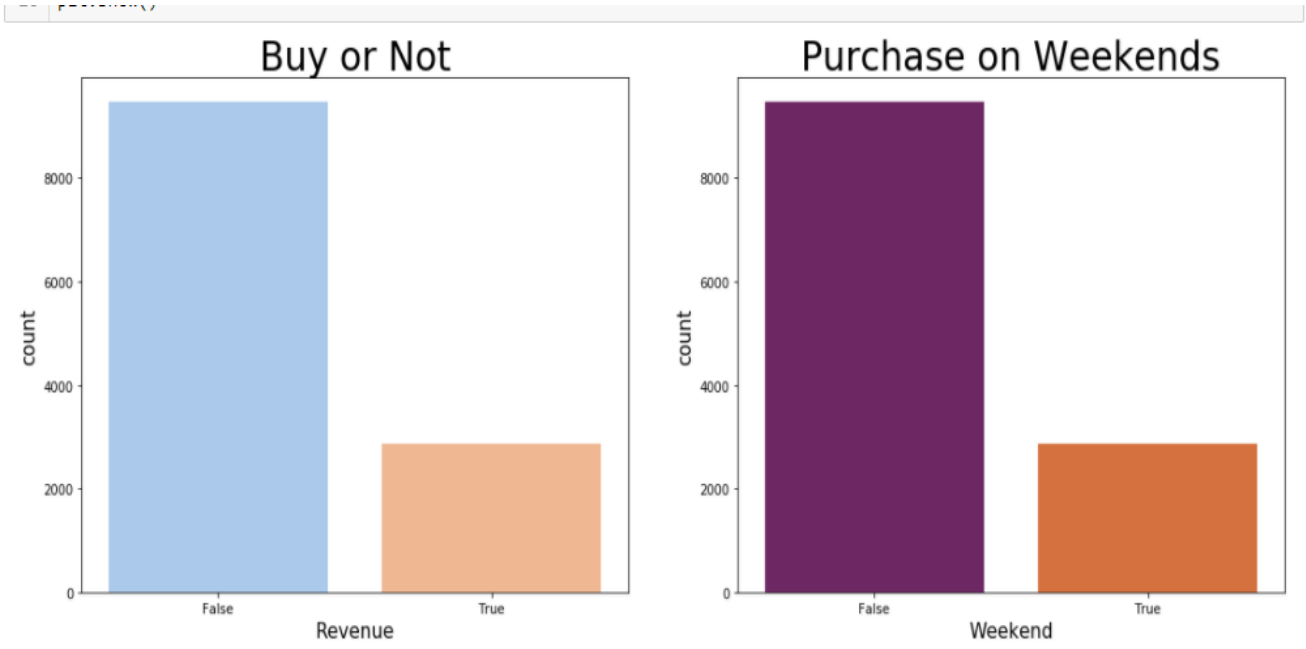
w_{i}(new)\:=\:w_{i}(old)

b(new)\:=\:b(old)

Here **'y'** is the actual output and **'t'** is the desired/target output.

Step 8 − Test for the stopping condition, which would happen when there is no change in weight.

Graph for buy or not and purchase on weekends



Section:KM101

Program

```
import numpy as np
from src.NeuralNetwork import NeuralNetwork
import src.utils as utils

def main():
    # ====================================
    # Settings
    # ====================================
    csv_filename = "online_shoppers_intention.csv"
    hidden_layers = [1] # number of nodes in hidden layers i.e. [layer1, layer2, ...]
    eta = 0.1 # learning rate
    n_epochs = 100 # number of training epochs
    n_folds = 4 # number of folds for cross-validation
    seed_crossval = 1 # seed for cross-validation
    seed_weights = 1 # seed for NN weight initialization

    # ====================================
    # Read csv data + normalize features
    # ====================================
    print("Reading '{}'...".format(csv_filename))
    X, y, n_classes = utils.read_csv(csv_filename, target_name="y",
normalize=False)
    N, d = X.shape
    print(" -> X.shape = {}, y.shape = {}, n_classes = {}\n".format(X.shape,
y.shape, n_classes))

    print("Neural network model:")
    print(" input_dim = {}".format(d))
    print(" hidden_layers = {}".format(hidden_layers))
    print(" output_dim = {}".format(n_classes))
    print(" eta = {}".format(eta))
    print(" n_epochs = {}".format(n_epochs))
    print(" n_folds = {}".format(n_folds))
    print(" seed_crossval = {}".format(seed_crossval))
    print(" seed_weights = {}\n".format(seed_weights))

    # ====================================
```

Section:KM101

```
    # Create cross-validation folds
    # ====================================
    idx_all = np.arange(0, N)
    idx_folds = utils.crossval_folds(N, n_folds, seed=seed_crossval) # list of list of
fold indices

    # ====================================
    # Train/evaluate the model on each fold
    # ====================================
    acc_train, acc_valid = list(), list()  # training/test accuracy score
    print("Cross-validating with {} folds...".format(len(idx_folds)))
    for i, idx_valid in enumerate(idx_folds):

        # Collect training and test data from folds
        idx_train = np.delete(idx_all, idx_valid)
        X_train, y_train = X[idx_train], y[idx_train]
        X_valid, y_valid = X[idx_valid], y[idx_valid]

        # Build neural network classifier model and train
        model = NeuralNetwork(input_dim=d, output_dim=n_classes,
                              hidden_layers=hidden_layers, seed=seed_weights)
        model.train(X_train, y_train, eta=eta, n_epochs=n_epochs)

        # Make predictions for training and test data
        ypred_train = model.predict(X_train)
        ypred_valid = model.predict(X_valid)

        # Compute training/test accuracy score from predicted values
        acc_train.append(100*np.sum(y_train==ypred_train)/len(y_train))
        acc_valid.append(100*np.sum(y_valid==ypred_valid)/len(y_valid))

        # Print cross-validation result
        print(" Fold {}/{}: acc_train = {:.2f}%, acc_valid = {:.2f}% (n_train = {},
n_valid = {})".format(
            i+1, n_folds, acc_train[-1], acc_valid[-1], len(X_train), len(X_valid)))

    # ====================================
    # Print results
    # ====================================
    print("  -> acc_train_avg = {:.2f}%, acc_valid_avg = {:.2f}%".format(
```

Section:KM101

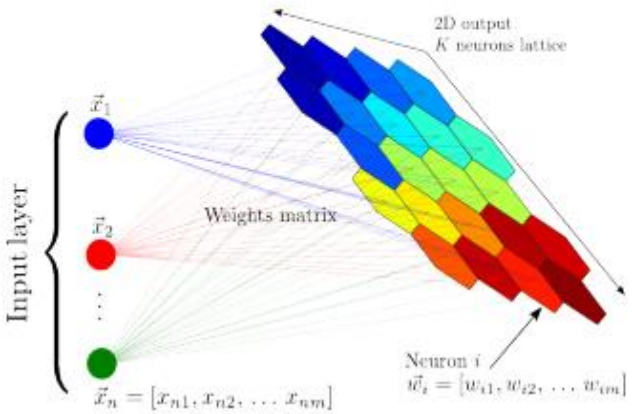
```
        sum(acc_train)/float(len(acc_train)), sum(acc_valid)/float(len(acc_valid))))

# Driver
if __name__ == "__main__":
    main()
```

Self-Organizing Map

Self-organizing maps are a class of unsupervised learning neural networks used for feature detection. They're used to produce a low-dimension space of training samples. Therefore, they're used for dimensionality reduction.

SOMs differ from other artificial neural networks because they apply competitive learning as opposed to error correlated learning, which involves backpropagation and gradient descent. In competitive learning, nodes compete for the right to respond to the input data subset. The training data usually has no labels and the map learns to differentiate and distinguish features based on similarities.



Section:KM101

Components of Self Organization

- **Initialization**: all connection weights are initialized to random values.
- **Competition**: Output nodes compete against themselves to be activated. Only one of them is activated at a time. The activated neuron is called a winner task all *neuron*. Because of this competition, the neurons are forced to organize themselves, forming a self organizing map *(SOM).*
- **Cooperation**: The spatial location of a topological neighborhood of excited neurons is determined by the winning neuron. This provides the basis for cooperation among neighboring neurons.
- **Adaptation**: Excited neurons decrease individual values of the discriminant function. This is done in relation to the input pattern via suitable adjustment of the associated connection weights. This way, the response of the winning neuron to the subsequent application of a similar input pattern is enhanced. The discriminant function is defined as the squared Euclidean distance between the input vector x and the weight vector wj for each neuron j:

$$d_j(\mathbf{x}) = \sum_{i=1}^{D} (x_i - w_{ji})^2$$

Steps for training a Self-Organizing Map

Training a self-organizing map occurs in several steps:

1. Initialize the weights for each node. The weights are set to small standardized random values.

2. Choose a vector at random from the training set and present to the lattice.

Section:KM101

3. Examine every node to calculate which one's weight is most like the input vector. This will allow you to obtain the Best Matching Unit (BMU). We compute the BMU by iterating over all the nodes and calculating the Euclidean distance between each node's weight and the current input vector. The node with a weight vector closest to the input vector is marked as the BMU.

4. Calculate the radius of the neighborhood of the BMU. Nodes found within the radius are deemed to be inside the neighborhood of the BMU.

5. Weights of the nodes found in step 4 are adjusted to make them more like the input vector. The weights of the nodes closer to the BMU are adjusted more.

6. Repeat step 2 for N iterations.

The conventional SOM learning algorithm can be explained using the following steps:

(a)Initialize the weight vectors *wi's* of the $m \times n$ neurons.

(b)
Randomly select an input vector $x(t)$and it is input to all the neurons at the same time in parallel.

(c)
Find the winner neuron *c*, i.e., BMU using the following equation:(1)c=argmin1⩽i⩽mn{‖wi(t)-x(t)‖},‖.‖ is the Euclidean distance measure. Where x(t) and wi(t) are the input and weight vector of neuron i at iteration *t* respectively.

(d)
The weight vector of the neurons is updated using the following equation:(2)wi(t+1)=wi(t)+hc,i(t)[x(t)-wi(t)],where hc,i(t) is a Gaussian neighborhood function [16] given below:(3)hc,i(t)=α(t)·exp-‖rc-ri‖22σ2(t),where *r* is the coordinate position of the neuron on the map, α(t) is the learning rate and σ(t) is the width of neighborhood radius.
Both α(t) and σ(t) decrease monotonically using the following equation:(4)α(t)=α(0)α(T)α(0)t/T,(5)σ(t)=σ(0)σ(T)σ(0)t/Twhere *T* is the training length.

Section:KM101

(e)
For all the input data, steps (b) to (d) are repeated.

Modified SOM

For each input data, the neurons at minimum and maximum distance from among 1-neighborhood of the BMU are found out as. These are then named nearest and farthest neuron for that particular input. The proposed learning algorithm of SOM can be summarized in the following steps:

(Step 1) All the weight vectors wi∈M of $m \times n$ neurons are initialized, where $i = 1, 2, \ldots, mn$ and M is a set of $m * n$ weight vectors. Then the winning frequency ηi=0 is initialized for all neurons and the connection value $C(i,j) = 0$ is also initialized between each neuron.

(Step 2) An input vector $x(t)$ is selected randomly and given simultaneously to all the neurons.

(Step 3) The winner neuron c, i.e., BMU is found out using Then, the distance between input x(t) and weight vector is found and the rank *ranki* is assigned to each neuron, where $i = 0, 1, \ldots mn$. The rank *ranki* is taken to be 0 for the BMU, because of being nearest to the input vector. The winning frequency ηc of the winner neuron c is increased by 1.

(Step 4) The farthest neuron and the nearest neuron are found out from among the 1-neighborhood of BMU using Euclidean equation.

(Step 5) The connection value between BMU and neuron i is increased using the following equation:

(6)C(c,i)=C(c,i)+1,where $i = f$ or $i \in Sf$.

Section:KM101

PROGRAM

```
import pandas as pd
import  numpy as np
from sklearn.metrics import accuracy_score
df =pd.read_csv('C:\\Users\\admin\\Desktop\\online_shoppers_intention.csv')
int11=np.column_stack((df.ProductRelated,df.ProductRelated_Duration,df.Bounce
Rates,df.ExitRates,df.OperatingSystems))
y=df['Revenue']
from sklearn import preprocessing
min_max_scaler=preprocessing.MinMaxScaler()
x_scaled=min_max_scaler.fit_transform(int11)
int1=pd.DataFrame(x_scaled)
def Classification(w_,int2):
      t_pre=[]
      for i in range(int2.shape[0]):
            temp=[]
            for j in range(w_.shape[1]):
               sum=0
               for k in range(w_.shape[0]):
                     sum=sum+((int2[i][k]-w_[k][j])**2)
               temp.append(sum)
            if(temp[0]<temp[1]):
               t_pre.append('True')
            else:
               t_pre.append('False')
      return t_pre
def KSOM(int1,revenue):
   t=[]
   X=np.array(int1[:5000])
   y=np.array(revenue[:5000])
   W_=np.random.uniform(low=0,high=1,size=(X.shape[1],2))
   lr=0.6
   EPOCH=1
   for iter in range(EPOCH):
       for i in range(X.shape[0]):
            temp=[]
            print(" for input ",X[i,:])
```

Section:KM101

```
            for j in range(W_.shape[1]):
                sum=0
                for k in range(W_.shape[0]):
                    sum=sum+((X[i][k]-W_[k][j])**2)

                temp.append(sum)
            if(temp[0]<temp[1]):
                index=0
            else:
                index=1
            print("neuron at position ",index," won")
            for p in range(W_.shape[0]):
                W_[p][index]=W_[p][index] + lr*(X[i][p]-W_[p][index])
            print(W_)
        lr=0.5*l
 t_pr=Classification(W_,X)
    for i in range(5000):
        t.append(str(y[i]))
    print('Accuracy:%.2f'%accuracy_score(t,t_pr))
KSOM(int1,y)
```

Section:KM101

SUPPORT VECTOR MACHINE

In machine learning**,support-vector machines** (**SVMs**, also **support-vector networks**) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

When data are unlabelled, supervised learning is not possible, and an unsupervised learning approach is required, which attempts to find natural clustering of the data to groups, and then map new data to these formed groups. The **support-vector clustering** algorithm, created by Hava Siegelmann and Vladimir Vapnik applies the statistics of support vectors, developed in the support vector machines algorithm, to categorize unlabeled data, and is one of the most widely used clustering algorithms in industrial applications.

How Svm classifier Works?

For a dataset consisting of features set and labels set, an SVM classifier builds a model to predict classes for new examples. It assigns new example/data points to one of the classes. If there are only 2 classes then it can be called as a Binary SVM Classifier.

There are 2 kinds of SVM classifiers:

1. **Linear SVM Classifier**
2. **Non-Linear SVM Classifier**

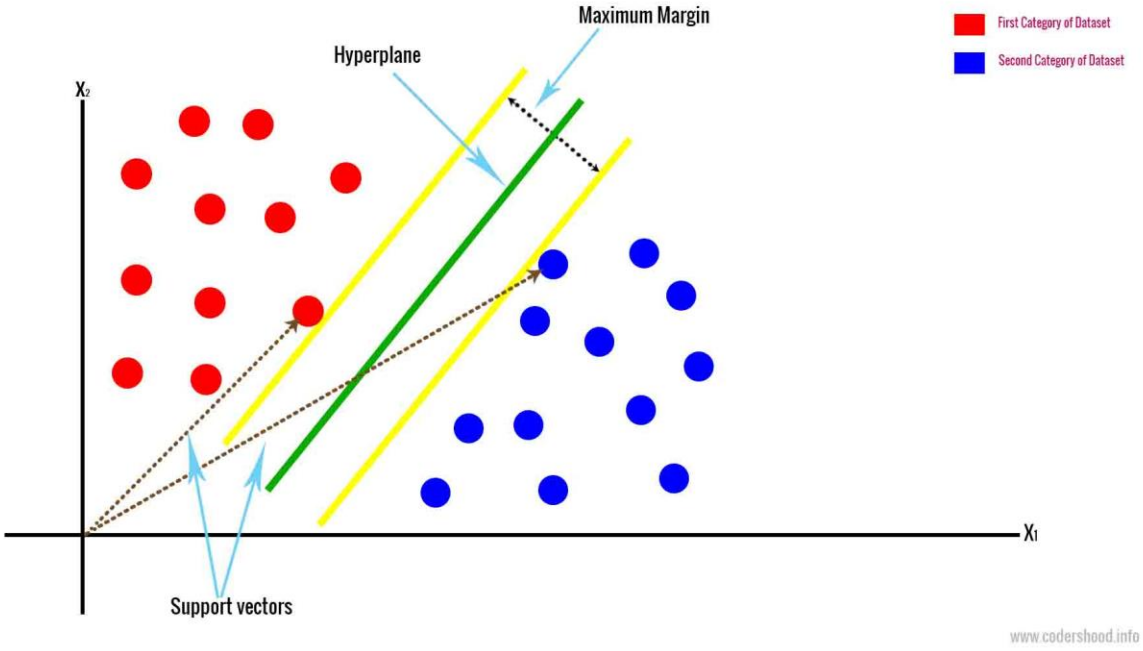
1.)**Svm Linear Classifier:**

In the linear classifier model, we assumed that training examples plotted in space. These data points are expected to be separated by an apparent gap. It predicts a straight hyperplane dividing 2 classes. The primary focus while drawing the

hyperplane is on maximizing the distance from hyperplane to the nearest data point of either class. The drawn hyperplane called as a maximum-margin hyperplane.

2.)**SVM Non-Linear Classifier:**

In the real world, our dataset is generally dispersed up to some extent. To solve this problem separation of data into different classes on the basis of a straight linear hyperplane can't be considered a good choice. For this Vapnik suggested creating Non-Linear Classifiers by applying the kernel trick to maximum-margin hyperplanes. In Non-Linear SVM Classification, data points plotted in a higher dimensional space.



Linear Support Vector Machine Classifier:

In Linear Classifier, A data point considered as a p-dimensional vector(list of p-numbers) and we separate points using (p-1) dimensional hyperplane. There can be many hyperplanes separating data in a linear order, but the best hyperplane is considered to be the one which maximizes the margin i.e., the distance between hyperplane and closest data point of either class.

The Maximum-margin hyperplane is determined by the data points that lie nearest to it. Since we have to maximize the distance between hyperplane and the data points. These data points which influences our hyperplane are known as support vectors.

Section:KM101

Non-Linear Support Vector Machine Classifier

Vapnik proposed Non-Linear Classifiers in 1992. It often happens that our data points are not linearly separable in a p-dimensional(finite) space. To solve this, it was proposed to map p-dimensional space into a much higher dimensional space. We can draw customized/non-linear hyperplanes using Kernel trick.
Every kernel holds a non-linear kernel function.

This function helps to build a high dimensional feature space. There are many kernels that have been developed. Some standard kernels are:

1.)Polynomial Kernal

It is popular in image processing.
Equation is:

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$$

where d is the degree of the polynomial.

2.) Gaussian radial basis function (RBF)

It is a general-purpose kernel; used when there is no prior knowledge about the data. Equation is:

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$$

, for: $\gamma > 0$

Sometimes parametrized using:

$$\gamma = 1/2\sigma^2$$

3.) Sigmoid kernel

We can use it as the proxy for neural networks. Equation is:

$$k(x, y) = \tanh(\alpha x^T y + c)$$

Advantages of SVM Classifier:

1.)SVMs are effective when the number of features is quite large.

2.)It works effectively even if the number of features are greater than the number of samples.

3.)Non-Linear data can also be classified using customized hyperplanes built by using kernel trick.

Disadvantages of SVM Classifier:

1.)The biggest limitation of Support Vector Machine is the choice of the kernel. The wrong choice of the kernel can lead to an increase in error percentage.

2.)With a greater number of samples, it starts giving poor performances.

3.)SVMs have good generalization performance but they can be extremely slow in the test phase.

4.)SVMs have high algorithmic complexity and extensive memory requirements due to the use of quadratic programming.

PROGRAM

```
import sklearn
import csv
import pandas as pd
import numpy as np
from sklearn.svm import SVC
data=pd.read_csv("E:\\online_shoppers_intention.csv")
print(data.shape[1])
print(data.shape[0])
print(data.shape)
data1=np.array(data)
y=data1[:,12]# column
z=list(y)
x=np.column_stack((data.Weekend,data.Revenue,data.Browser,data.Region,data.T
rafficType,data.Administrative_Duration))
#independent features
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,z,test_size=0.5,random_state=0)
from sklearn.svm import SVC #support vector classifier
svm=SVC(kernel='linear',random_state=0)
svm.fit(x_train,y_train)
```

Section:KM101

```
y_pred=svm.predict(x_test)#classification
print('misclassified samples : %d'%(y_test!=y_pred).sum()) #compute
from sklearn.metrics import accuracy_score
print('accuracy :%.2f'%accuracy_score(y_test,y_pred))
```

O/P

```
18
12330
(12330, 18)
misclassified samples : 0
accuracy :1.00
```

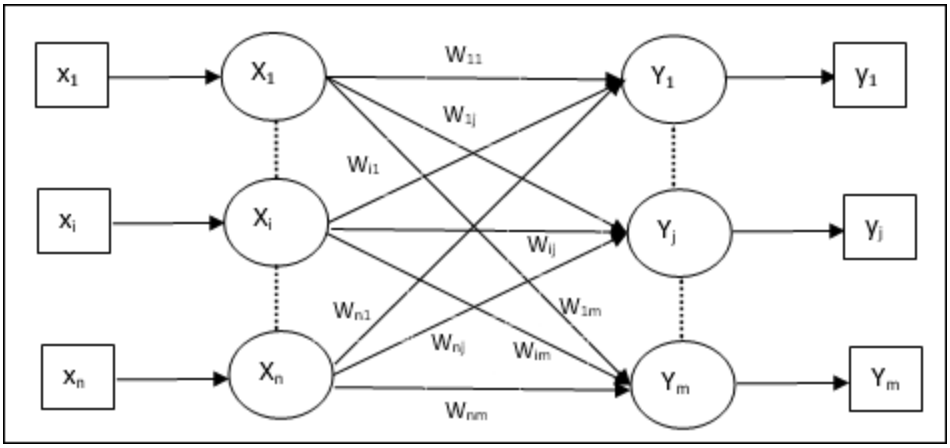
Section:KM101

Learning Vector Quantitation: -

Learning Vector Quantization LVQ which uses supervised learning. We may define it as a process of classifying the patterns where each output unit represents a class. As it uses supervised learning, the network will be given a set of training patterns with known classification along with an initial distribution of the output class. After completing the training process, LVQ will classify an input vector by assigning it to the same class as that of the output unit.

Architecture: -

Following figure shows the architecture of LVQ which is quite similar to the architecture of KSOM. As we can see, there are **"n"** number of input units and **"m"** number of output units. The layers are fully interconnected with having weights on them.



Parameters Used: -

Following are the parameters used in LVQ training process as well as in the flowchart

- **x** = training vector $(x_1,...,x_i,...,x_n)$.
- **T** = class for training vector **x.**
- $\mathbf{w_j}$ = weight vector for $\mathbf{j^{th}}$ output unit.
- $\mathbf{C_j}$ = class associated with the $\mathbf{j^{th}}$ output unit.

Section:KM101

Training Algorithm: -

Step 1 − Initialize reference vectors, which can be done as follows −

- **Step 1**a − From the given set of training vectors, take the first "**m**" number of clusters training vectors and use them as weight vectors. The remaining vectors can be used for training.
- **Step 1**b − Assign the initial weight and classification randomly.
- **Step 1**c − Apply K-means clustering method.

Step 2 − Initialize reference vector αα

Step 3 − Continue with steps 4-9, if the condition for stopping this algorithm is not met.

Step 4 − Follow steps 5-6 for every training input vector **x**.

Step 5 − Calculate Square of Euclidean Distance for **j = 1 to m** and **i = 1 to n**

$$D(j)=\sum i=1n\sum j=1m(xi-wij)2D(j)=\sum i=1n\sum j=1m(xi-wij)2$$

Step 6 − Obtain the winning unit **J** where **D**jj is minimum.

Step 7 − Calculate the new weight of the winning unit by the following relation −

if $\mathbf{T = C_j}$ then wj(new)=wj(old)+α[x−wj(old)] wj(new)=wj(old)+α[x−wj(old)]

if $\mathbf{T \neq C_j}$ then wj(new)=wj(old)−α[x−wj(old)] wj(new)=wj(old)−α[x−wj(old)]

Step 8 − Reduce the learning rate αα.

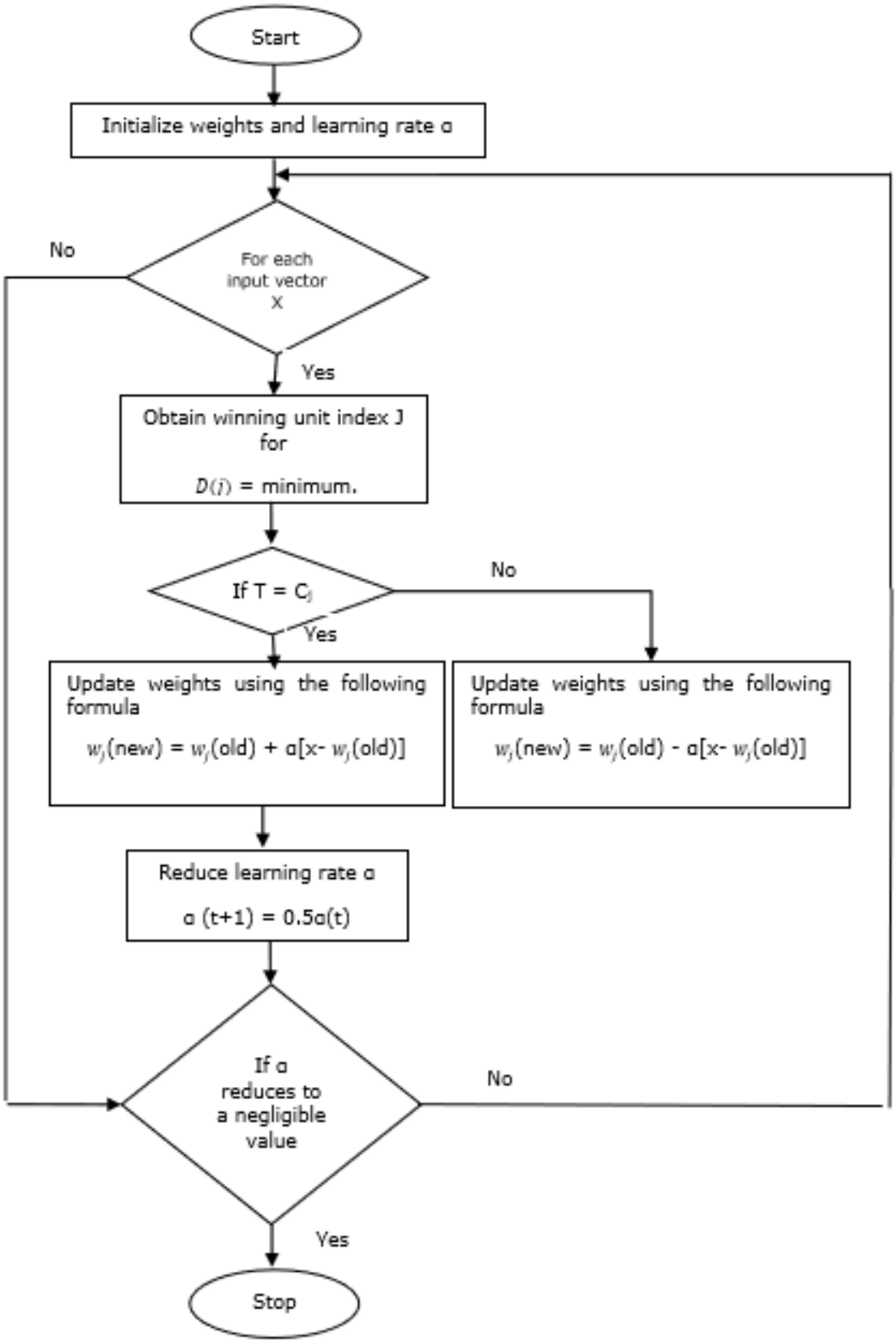
Step 9 − Test for the stopping condition. It may be as follows −

- Maximum number of epochs reached.
- Learning rate reduced to a negligible value.

Section:KM101

Flowchart: -

Section:KM101



Section:KM101

Variants: -

Three other variants namely LVQ2, LVQ2.1 and LVQ3 have been developed by Kohonen. Complexity in all these three variants, due to the concept that the winner as well as the runner-up unit will learn, is more than in LVQ.

Program: -

```
from math import sqrt
from random import randrange
from csv import reader
import numpy as np
import time

def Read_file(file_name):
    dataset = list()
    with open(file_name, 'r',encoding='utf-8') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())
```

Section:KM101

```
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup

def dataset_minmax(dataset):
    stats = [[min(column), max(column)] for column in zip(*dataset)]
    return stats

def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row) - 1):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

testdataset = Read_file('online_shoppers_intention.csv')

# change string column values to float
for i in range(len(testdataset[0]) - 1):
    str_column_to_int(testdataset, i)

    # # convert last column to integers
str_column_to_int(testdataset, len(testdataset[0]) - 1)
```

Section:KM101

```
minmax = dataset_minmax(testdataset)
normalize_dataset(testdataset, minmax)

def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

def get_best_matching_unit(codebooks, test_row):
    distances = list()
    for codebook in codebooks:
        dist = euclidean_distance(codebook, test_row)
        distances.append((codebook, dist))
    distances.sort(key=lambda tup: tup[1])
    return distances[0][0]

def random_codebook(train):
    n_records = len(train)
    n_features = len(train[0])
    codebook = [train[randrange(n_records)][i] for i in range(n_features)]
    return codebook

def train_codebooks(train, n_codebooks, lrate, epochs):
    codebooks = [random_codebook(train) for i in range(n_codebooks)]
    for epoch in range(epochs):
```

Section:KM101

```
        rate = lrate * (1.0-(epoch/float(epochs)))
        sum_error = 0.0
        for row in train:
            bmu = get_best_matching_unit(codebooks, row)
            for i in range(len(row)-1):
                error = row[i] - bmu[i]
                sum_error += error**2
                if bmu[-1] == row[-1]:
                    bmu[i] += rate * error
                else:
                    bmu[i] -= rate * error
    return codebooks

def predict(codebooks, test_row):
    bmu = get_best_matching_unit(codebooks, test_row)
    return bmu[-1]
def learning_vector_quantization(train, test, n_codebooks, lrate, epochs):
    codebooks = train_codebooks(train, n_codebooks, lrate, epochs)
    predictions = list()
    for row in test:
        output = predict(codebooks, row)
        predictions.append(output)
    return(predictions)

start_time= time.time()
test_set = testdataset
learn_rate = 0.5
```

Section:KM101

```
n_epochs = 30
n_codebooks = 20
runs = [0]*10
w = 0
r = 1
for i in range(len(runs)):
    r = r + 100
    if (r + 200) > 699:
        r = w
        w += 30
    train_set = [test_set[i] for i in range(r, r + 200)]
    predicted = learning_vector_quantization(train_set, test_set, n_codebooks,learn_rate,n_epochs)
    total = 0
    correct = 2
    for row in test_set:
        actual = row[-1]
        if actual == predicted[total]: correct +=1
        total += 1
    accuracy = correct*100/total
    print('Learning_Rate: {}  n_epochs: {} n_codebooks = {} Accuracy: {}' .format(learn_rate,n_epochs,n_codebooks,accuracy))


    runs[i] = accuracy


mean = sum(runs)/len(runs)
print("Mean_Accuracy: {}".format(mean))
```

Section:KM101

Output: -

Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.413267374908
77
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.421377017273
54
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.267293812342
87
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.534912010380
34
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.534912010380
34
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.534912010380
34
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.267293812342
87
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.186197388695
16
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.526802368015
57
Learning_Rate: 0.5 n_epochs: 30 n_codebooks = 20 Accuracy: 84.534912010380
34
Mean_Accuracy: 84.42218798151

Section:KM101

Section:KM101