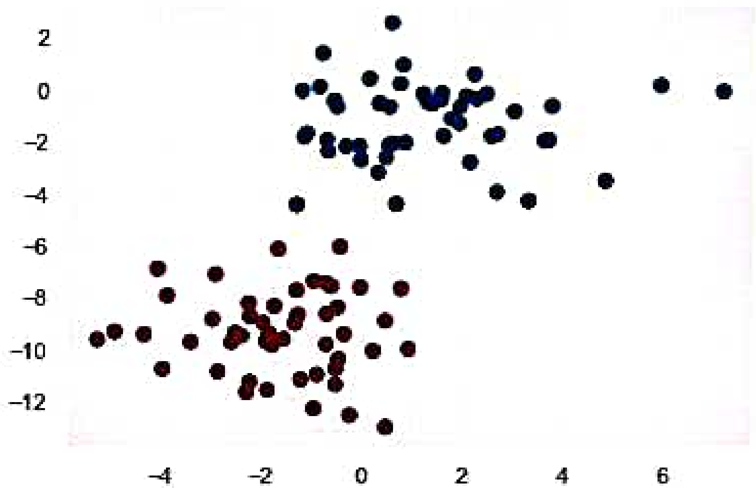
In [6]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
```

Out[6]: <matplotlib.collections.PathCollection at 0x222116037f0>



Experiment 7
Example 1:

Implement Naive Bayes theorem to classify the english text.

Source Code:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs
x,y = make_blobs(100,2,centers=2,random_state=2,
                          cluster_std=1.5)
plt.scatter(x[:,0], x[:,1], c=y, s=50, cmap='RdBu').
```

localhost:8889/notebooks/Untitled3.ipynb

azon.in AliExpress

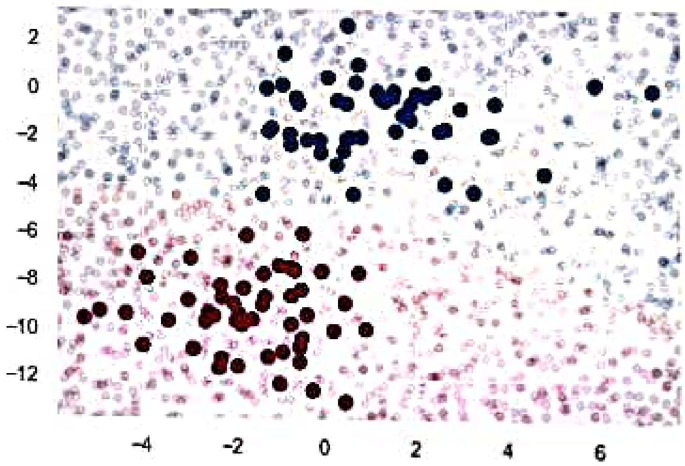
jupyter Untitled3 Last Checkpoint: 10 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run Code

```
yprob = model.predict_proba(Xnew)
yprob[-8:].round(2)
```

```
Out[7]: array([[0.89, 0.11],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [1.  , 0.  ],
               [0.  , 1.  ],
               [0.15, 0.85]])
```



Example 2:

Implement Naive Bayes theorem to classify the english text

Source code:

```
% matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs
x,y = make_blobs (100, 2, centers =2, random_state=2, cluster_
                                    std = 1.5)
plt.scatter (x[:,0], x[:,1], c=y, s=50, cmap='RdBu');
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit (x,y);
rng = np.random.RandomState(0)
Xnew = [-6,-14] + [14,18]* rng.rand (2000,2)
Ynew = model.predict (Xnew)
plt.scatter (x[:,0], x[:,1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter (Xnew[:,0], Xnew[:,1], c=ynew, s=20, cmap='RdBu',
                                    alpha =0.1)
plt.axis(lim);
yprob = model.predict_proba (Xnew)
yprob[-8:].round (2)
```

localhost:8889/notebooks/Untitled3.ipynb

AliExpress

ter Untitled3 Last Checkpoint: 12 minutes ago (unsaved changes)

dit View Insert Cell Kernel Widgets Help

Run Code

```
# define the population size
n_pop = 100
# crossover rate
r_cross = 0.9
# mutation rate
r_mut = 1.0 / float(n_bits)
# perform the genetic algorithm search
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

```
>0, new best f([1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1]) = -15.000
>1, new best f([1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]) = -16.000
>3, new best f([1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -17.000
>4, new best f([1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -18.000
>4, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000
>7, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
```

[ ]:

Experiment 8:

//genetic algorithm to evaluates a binary string based on the number of 1's in the string. a bitstring with a length of 20 bits will have a score of 20 for a string of all 1's in the string.
11111111111111111111 = 20, 11111111110000000000 = 10)

```
from numpy.random import randint
from numpy.random import rand
# objective function
def onemax (x):
return - sum(x)
# tournament selection
def selection (pop, scores, k=3):
# first random selection
selection_ix= randit (len (pop))
for ix in randint (0,len(pop),k-1):
# checks if better (e.g. perform a tournament)
if scores [ix] < scores [selection_ix]:
    selection_ix = ix
    return pop[selection_ix]
# crossover two parents to create two children
def crossover (p1 ,p2 , r_cross):
# children are copies of parents by default
    c1 ,c2 = p1.copy() ,p2.copy()
# check for recombination
    if rand() < r_cross:
```

```
# select crossover point that is not on the end of the
  string.
  pt = randint(1, len(p1) - 2)
# perform crossover
    c1 = p1[:pt] + p2[pt:]
    c2 = p2[:pt] + p1[pt:]
    return [c1, c2]
  # mutation operator
  def mutation(bitstring r_mut):
    for i in range(len(bitstring)):
  # check for a mutation
    if rand() < r_mut:
    # flip the bit
    bitstring[i] = 1 - bitstring[i]
#genetic algorithm
 def genetic_algorithm(objective, n-bits, n-iter, n-pop,
                        r-cross, r-mut):
# initial population of random bitstring
  pop = [randit(0, 2, n-bits). tolist() for _ in range(n-pop)]
# keep track of best solution
      best, best_eval = 0,
        objective(pop[0])
  # enumerate generations
   for gen in range(n_iter):
  # evaluate all candidates in the population
    scores = [objective(c) for c in pop]
```

```
# check for new best solution
    for i in range (n_pop):
    if & scores [i] < best_eval:
    best, best_eval = pop[i];
    scores[i]
Print(">%d, new best f(%s) = %.3f" % (gen, pop[i], scores[i])
# select parents
selected = [selection (pop, scores) for_ in range (n_pop)]
# create the next generation
    children = list()
    for i in range (0, n_pop, 2):
# get selected parents in pairs
    P1, P2 = selected [i], selected [i+1]
# crossover and mutation
    for c in crossover (P1, P2, r_cross):
# mutation
    mutation (c, r_mut)
# store for next generation
        children.append (c)
# replace population
        pop = children
    return [best, best_eval]
# define the total iterations
        n_iter = 100
# bits
    n_bits = 20
```

```
#define the population size
  n-pop =100
# crossover rate
r-cross = 0.9
# mutation rate
r-mut = 1.0 / float (n-bits)
#perform the genetic algorithm search
best , score = genetic - algorithm (one max, n-bits, n-iter,
                                   n-pop, r-cross, r-mut)
   Print ('Done !')
  Print ('f(%s) = %f' % (best, score))
```

localhost:8889/notebooks/Untitled3.ipynb

azon.in AliExpress

jupyter Untitled3 Last Checkpoint: 27 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run Code

```
>epoch=0, lrate=0.500, error=6.350
>epoch=1, lrate=0.500, error=5.531
>epoch=2, lrate=0.500, error=5.221
>epoch=3, lrate=0.500, error=4.951
>epoch=4, lrate=0.500, error=4.519
>epoch=5, lrate=0.500, error=4.173
>epoch=6, lrate=0.500, error=3.835
>epoch=7, lrate=0.500, error=3.506
>epoch=8, lrate=0.500, error=3.192
>epoch=9, lrate=0.500, error=2.898
>epoch=10, lrate=0.500, error=2.626
>epoch=11, lrate=0.500, error=2.377
>epoch=12, lrate=0.500, error=2.153
>epoch=13, lrate=0.500, error=1.953
>epoch=14, lrate=0.500, error=1.774
>epoch=15, lrate=0.500, error=1.614
>epoch=16, lrate=0.500, error=1.472
>epoch=17, lrate=0.500, error=1.346
>epoch=18, lrate=0.500, error=1.233
>epoch=19, lrate=0.500, error=1.132
[{'weights': [-1.4688375095432327, 1.850887325439514, 1.0858178629550297], 'output': 0.029980305604426185, 'delta': -0.00595466
04162323625}, {'weights': [0.37711098142462157, -0.0625909894552989, 0.2765123702642716], 'output': 0.9456229000211323, 'delt
a': 0.0026279652850863837}]
[{'weights': [2.515394649397849, -0.3391927502445985, -0.9671565426390275], 'output': 0.23648794202357587, 'delta': -0.04270059
278364587}, {'weights': [-2.5584149848484263, 1.0036422106209202, 0.4238308646758271 5], 'output': 0.7790535202438367, 'delta':
0.03803132596437354}]
```

Experiment 9:

Implement the finite words classification system using Back Propogation algorithm.

Source code:

```
from math import exp
from random import seed
from random import random
import matplotlib.pyplot as plt
#initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_Layer = [{'weights': [random() for i in range
                                  (n_inputs +1)]}
    for i in range (n_hidden)]
    network.append (hidden_Layer)
    output_layer = [{'weights': [random() for i in range
                                  (n_hidden+1)]}
    for i in range (n_outputs)]
    network.append (output_layer)
    return network
# Calculate neuron activation for an input
def activate (weights, inputs):
    activation = weights [-1]
    for i in range (len (weights) - 1):
    activation+ = weights [i] * inputs [i]
    return activation
```

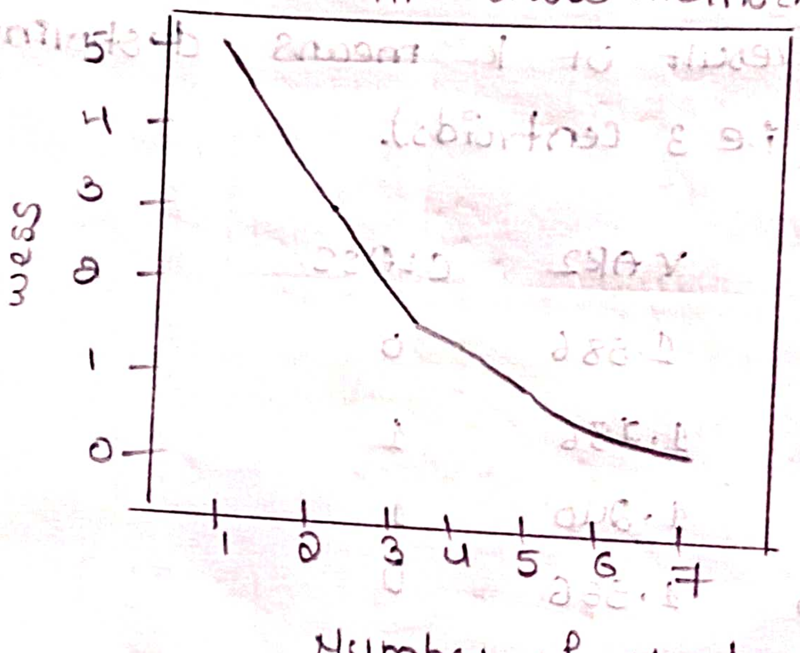
```
# Transfer neuron activation
def transfer (activation):
return 1.0 / (1.0 + exp(-activation))
# Forward propagate input to a network output
def forward_propagate (network, row):
        inputs = row
    for layer in network:
        new_inputs = []
    for neuron in layer:
   activation = activate (neuron ['weights'], inputs)
   neuron ['output'] = transfer (activation)
   new_inputs.append (neuron ['output'])
        inputs = new_inputs
       return inputs
# calculate the derivative of an neuron output
    def transfer_derivative (output):
        return output * (1.0 - output)
# Back propagate error and store in neurons
  def
  backward_propagate_error (network, expected):
  for i in reversed (range (len(network))):
       layer = network [i]
       errors = list()
       if i != len (network) - 1:
       for j in range (len (layer)):
              error = 0.0
```

```
for neuron in network [i+1]:
  error + = (neuron ['weights'] [j] * neuron['delta'])
   errors . append (error)
 else:
   for j in range (len (layer)):
    neuron = layer[j]
    neuron['delta'] = errors[j] *
  transfer_derivative (neuron['output'])
# update network weights with error
  def update - weights (network, row, l_rate):
   for i in range (len(network)):
      inputs = row [i-1]
      if i! = 0:
      inputs = [neuron['output'] for neuron in network [i-1]]
   for neuron in network [i]:
    for j in range (len(inputs)):
     neuron ['weights'] [j] + = l_rate * neuron ['delta'] * inputs[j]
  neuron['weights'] [-1] + = l_ rate *  neuron['delta']
 # Train a network (network, train, l_rate, n_epoch, n_outputs):
   for epoch in range (n_epoch):
       sum_ error = 0
      for row in train:
  outputs = forward - propagate (network, row)
    expected = [o for i in range (n_outputs)]
    expected [row[-1]] = 1
       sum_error + = sum ([(expected [i] - outputs [i] **2
```

```
for i in range (len (expected))])
backward - propagate_ error (network ,expected)
 update - weights (network , row , l-rate)
Print ('>epoch = %d, lrate = %.3f , error = %.3f' % (epoch ,
                                     l_rate , sum-error))
# Test training backprop algorithm seed (1)
 dataset = [[2.7810836 ,2.550537003,0],
           [1.465489372 , 2.362125076,0],
           [3.396561688, 4.400293529,0],
           [1.38807019 ,1.850220317,0],
           [3.06407232 , 3.005305973 ,0],
           [7.627531214 , 2.759262235 ,1],
           [5.332441248 ,2.088626775 ,1],
           [6.922596716 , 1.77106367,1],
           [8.675418651 , -0.242068655,1],
           [7.673756466, 3.508563011,1]]
 n_inputs = len(dataset[0]) - 1
 n_outputs = len ( set ( [row [-1] for row in dataset]))
 network = initialize _network ( n_inputs, 2, n_outputs)
 train _network ( network, dataset , 0.5 , 20 , n_outputs)
  for Layer in network:
     print (layer)
```

O/P [5.631208500000005, 2.487442083333333, 1.385325, 0.8182311666666667, 0.55849783333334, 0.2542455000000005, 0.17530750000000008]

The elbow method



O/P :

KMeans(n - clusters = 3, random = state - 0)

[0 1 1 1 2 0 2 1 2 2]

Experiment - 4

Given the following data, which specify classifications for nine combinations of VAR1 and VAR2 predict a classification for a case where VAR1 = 0.906 and VAR2 = 0.606, using the result of k-means clustering with 3 means (i.e., 3 centroids)

VAR 1	VAR 2	CLASS
1.713	1.586	0
0.180	1.786	1
0.353	1.240	1
0.940	1.566	0
1.486	0.759	1
1.266	1.106	0
1.540	0.419	1
0.459	1.799	1
0.773	0.186	1

Source code:

```
# importing the libraries
    import numpy as np
    import matplotlib.PyPlot as Plt
    import pandas as pd
# importing the Iris dataset with pandas
  dataset = Pd.read_csv[r'c:|users| Desktop|DL_data.csv')
  Print (dataset)
```

```
X = dataset . iloc [i, (1,2))] values
print (x)
# finding the optimum number of clusters for k means
classification
from Sklearn _ cluster import kMeans
wess = [ ]
for i in range (1,8);
Kmeans = kmeans (n_ clusters = i, init = k_means ++;
            max_iter = 300, n_init =10, random_state =0)
kmeans fit (x)
wess. opened (kmeans . inertia)
Print ( wess)
# plotting the results onto a line graph, allowing us to
observe 'The elbow'
plt. plot (range (1,8), wess)
plt . tittle ('The elbow method')
plt xlabel ('Number of clusters)
plt . ylabel ('Wess')
plt . show( )
```