OIST Course B26: Introduction to Neuroscience

Reinforcement Learning and Supervised Learning

2019.12.12

In [18]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Reinforcement Learning

Classes for minimum environment and agent

In [19]:

```
class Environment:
    """Class for a reinforcement learning environment"""

    def __init__(self, nstate=3, naction=2):
        """Create a new environment"""
        self.Ns = nstate   # number of states
        self.Na = naction  # number of actions

    def start(self):
        """start an episode"""
        # randomly pick a state
        self.state = np.random.randint(self.Ns)
        return self.state

    def step(self, action):
        """step forward given an action"""
        # random reward
        self.reward = np.random.random()  # between 0 and 1
        # random state transition
        self.state = np.random.randint(self.Ns)
        return self.reward, self.state
```

In [20]:

```
class Agent:
    """Class for a reinforcement learning agent"""

    def __init__(self, nstate, naction):
        """Create a new agent"""
        self.Ns = nstate   # number of states
        self.Na = naction  # number of actions

    def start(self, state):
        """first action, without reward feedback"""
        # randomly pick an action
        self.action = np.random.randint(self.Na)
        return self.action

    def step(self, reward, state):
        """learn by reward and take an action"""
        # do nothing for reward
        # randomly pick an action
        self.action = np.random.randint(self.Na)
        return self.action
```

In [21]:

```
class RL:
    """Reinforcement learning by interacton of Environment and Agent"""

    def __init__(self, environment, agent, nstate, naction):
        """Create the environment and the agent"""
        self.env = environment(nstate, naction)
        self.agent = agent(nstate, naction)

    def episode(self, tmax=50):
        """One episode"""
        # First contact
        state = self.env.start()
        action = self.agent.start(state)
        # Table of t, r, s, a
        Trsa = np.zeros((tmax+1,4))
        Trsa[0,:] = [0, 0, state, action]
        # Repeat interactoin
        for t in range(1, tmax+1):
            reward, state = self.env.step(action)
            action = self.agent.step(reward, state)
            Trsa[t,:] = [t, reward, state, action]
        return Trsa

    def run(self, nrun=10, tmax=50):
        """Multiple runs of episodes"""
        Return = np.zeros(nrun)
        for n in range(nrun):
            r = self.episode(tmax)[:,1]  # reward sequence
            Return[n] = sum(r)
        return Return
```

Q learning of Pain-Gain task

In [22]:

```
class PainGain(Environment):
    """Pain-Gain environment """

    def __init__(self, nstate=4, naction=2, gain=6):
        super().__init__(nstate, naction)
        # setup the reward function as an array
        self.R = np.ones((self.Ns, self.Na))
        self.R[:,1] = -1   # small pains for action 1
        self.R[0,0] = -gain  # large pain
        self.R[-1,1] = gain  # large gain

    def step(self, action):
        """step by an action"""
        self.reward = self.R[self.state, action]  # reward
        self.state = (self.state + 2*action-1)%self.Ns  # move left or right
        return(self.reward, self.state)
```

In [23]:

```python
class QL(Agent):
    """Class for a Q-learning agent"""

    def __init__(self, nstate, naction):
        super().__init__(nstate, naction)
        # allocate Q table
        self.Q = np.zeros((nstate, naction))
        # default parameters
        self.alpha = 0.1  # learning rate
        self.beta = 2.0   # inverse temperature
        self.gamma = 0.9  # discount factor

    def boltzmann(self, q):
        """Boltzmann selection"""
        pa = np.exp( self.beta*q)   # unnormalized probability
        pa = pa/sum(pa)    # normalize
        return np.random.choice(self.Na, p=pa)

    def start(self, state):
        """first action, without reward feedback"""
        # Boltzmann action selection
        self.action = self.boltzmann( self.Q[state,:])
        # remember the state
        self.state = state
        return self.action

    def step(self, reward, state):
        """learn by reward and take an action"""
        # TD error: self.state/action retains the previous ones
        delta = reward + self.gamma*max(self.Q[state,:]) - self.Q[self.state,sel
        # Update the value for previous state and action
        self.Q[self.state,self.action] += self.alpha*delta
        # Boltzmann action selection
        self.action = self.boltzmann( self.Q[state,:])
        # remember the state
        self.state = state
        return self.action
```

Setup and Run

In [24]:

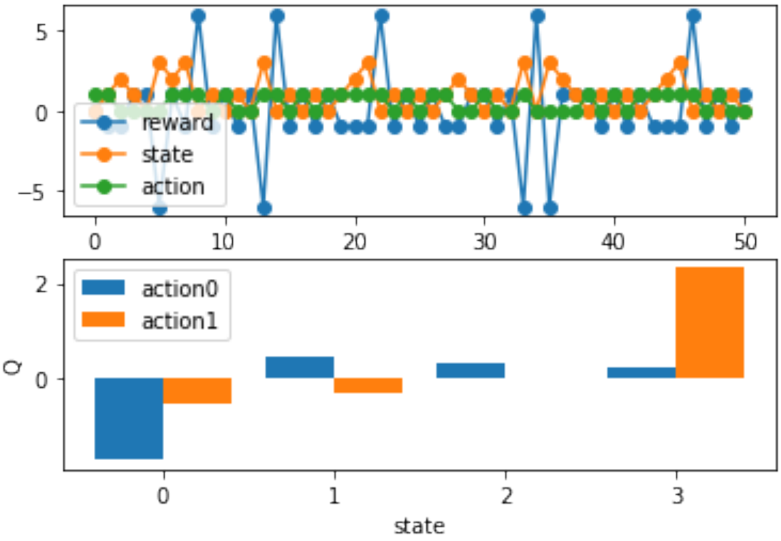
```python
# Setup Pain-Gain environment and Q-learning agent
pgq = RL(PainGain, QL, 4, 2)
pgq.env.R
```

Out[24]:

```
array([[-6., -1.],
       [ 1., -1.],
       [ 1., -1.],
       [ 1.,  6.]])
```

In [25]:

```python
# Run an episode of 50 trials
trsa = pgq.episode(50)
# plot the time course of state, action, reward,
plt.subplot(2,1,1)
plt.plot(trsa[:,1:], "o-")
plt.legend(["reward", "state", "action"])
plt.xlabel("time")
# plot the action values
plt.subplot(2,1,2)
plt.bar(np.arange(pgq.agent.Ns)-0.2, pgq.agent.Q[:,0], 0.4)  # action 0
plt.bar(np.arange(pgq.agent.Ns)+0.2, pgq.agent.Q[:,1], 0.4)  # action 1
plt.legend(["action0", "action1"]);
plt.xticks(range(pgq.agent.Ns)); plt.xlabel("state"); plt.ylabel("Q");
```



In [26]:

```python
# reduce the discount factor
pgq.agent.gamma = 0.3
```

In [27]:

```python
# restore the discount factor
pgq.agent.gamma = 0.9
```

In [28]:

```python
# increase the inverse temperature
pgq.agent.beta = 10
```

In [29]:

```python
# reduce the inverse temperature
pgq.agent.beta = 1
```

In [ ]:

```python

```

Unsupervised Learning

As we grow, we learn that there are different things and creatures in the world, such as plants and animals, and in more detail, dogs, cats and humans. What is remarkable is that most of such learning is done spontaneously without explicit teaching about what is a dog, or labels specifying which is a dog or which is cat. Learning categories without explicit labels is an example of *unspervised learning*. But how can we define categories without category labels?

The key in unsupervised learning is to find a certain structure in the distribution $p(\boldsymbol{x})$ that produced the observed data $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$.

Typical cases are:

- Dividing into clusters:
    - k-means clustering
    - mixtures of Gaussians
    - self-organizing maps (SOM)
- Decomposing into components:
    - principal component analysis (PCA)
    - singular value decomposition (SVD)
    - independent component analysis (ICA)

K-means Clustering

The most basic method of clustering is K*-means clustering*, which divides a data set $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ into K clusters.

We define prototypes $\boldsymbol{\mu}_k$ for $k = 1, \ldots, K$ clusters and specify belonging of data points by binary *indicator variables* $r_{nk} \in \{0, 1\}$.

A good clustering is achieved by minimizing the *distortion measure*

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2.$$

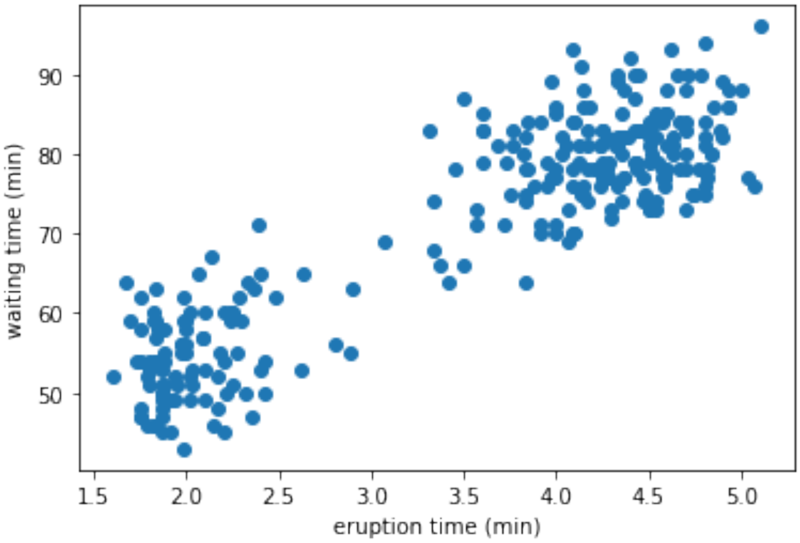
We do that by repeating the following steps:

- For the current prototypes $\boldsymbol{\mu}_k$, re-assign data points.
    - for each $\boldsymbol{x}_n$, find the nearest $\boldsymbol{\mu}_k$ and set $r_{nk} = 1$ and $r_{nj \neq k} = 0$.
- For the current assignment by r_{nk}, update the prototypes by

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} r_{nk} \boldsymbol{x}_n}{\sum_{n=1}^{N} r_{nk}}$$

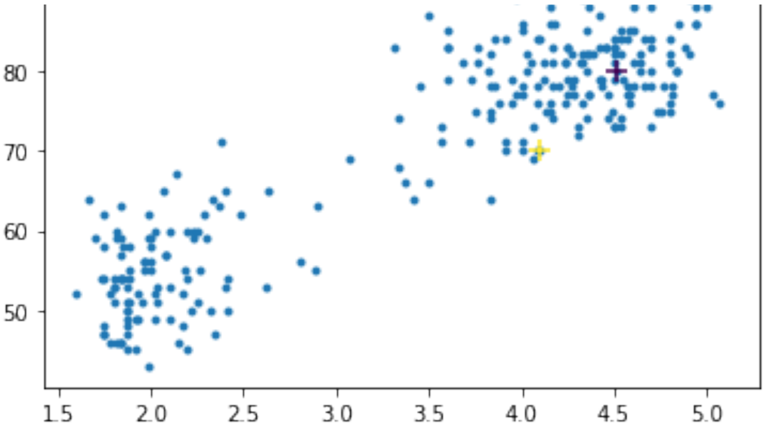
In [30]:

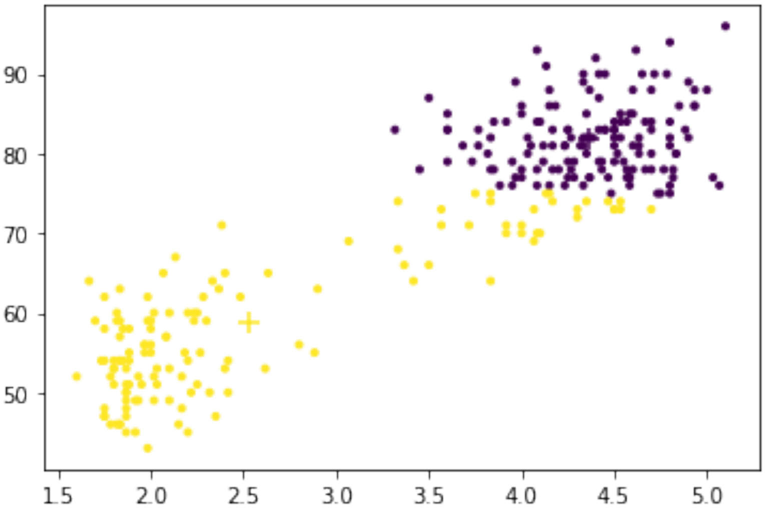
```
# Load data from a text file
X = np.loadtxt('faithful.txt')
N, D = X.shape
plt.scatter(X[:,0], X[:,1])
plt.xlabel('eruption time (min)')
plt.ylabel('waiting time (min)');
```

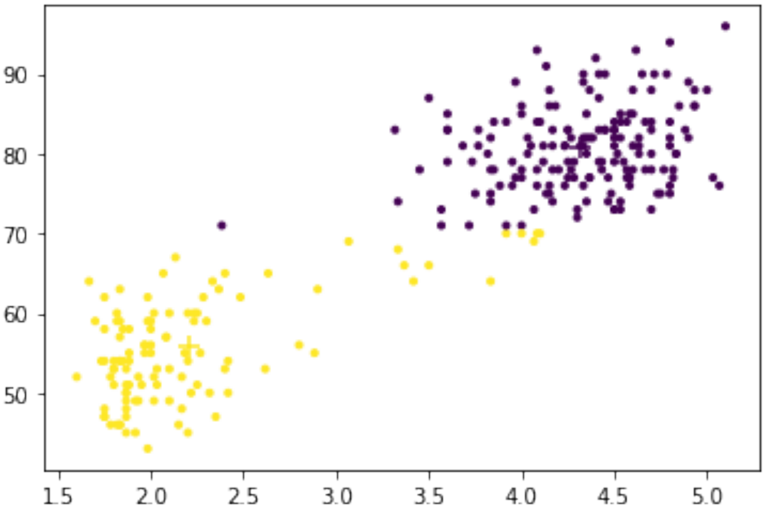


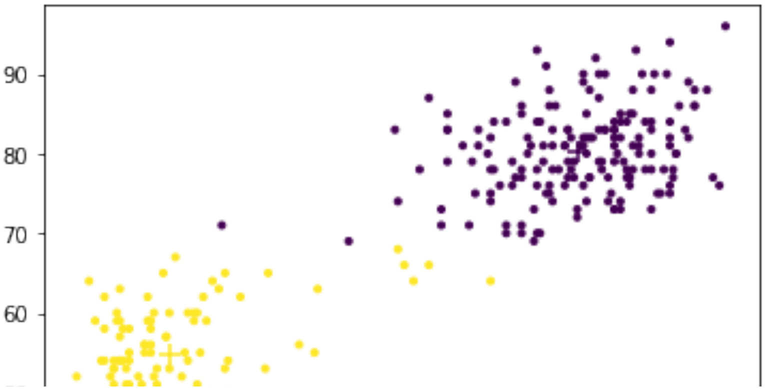
In [31]:

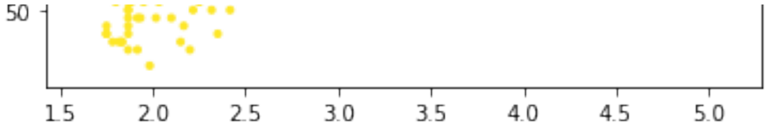
```
K = 2  # number of clusters
# Initial guess of prototypes
Mu = X[np.random.randint(0, N, K),:]   # pick K points randomly
plt.scatter(X[:,0], X[:,1], marker='.')
plt.scatter(Mu[:,0], Mu[:,1], c=range(K), marker='+', s=100)
plt.show()
Y = np.zeros(N, dtype=int)    # cluster label
R = np.zeros((N,K), dtype=bool)   # assignment matrix
for t in range(5):
    # Update assignment
    for n in range(N):
        # check the distances
        dist = [ np.dot(X[n]-Mu[k], X[n]-Mu[k]) for k in range(K)]
        # find the nearest mean
        Y[n] = np.argmin(dist)
        R[n,:] = np.zeros(K)
        R[n,Y[n]] = 1
    # show assignment
    plt.scatter(X[:,0], X[:,1], c=Y, marker='.')
    # Update the means
    for k in range(K):
        Mu[k] = np.mean(X[R[:,k]], axis=0)
    # plot the new means

    plt.scatter(Mu[:,0], Mu[:,1], c=range(K), marker='+', s=100)
    plt.show()
```

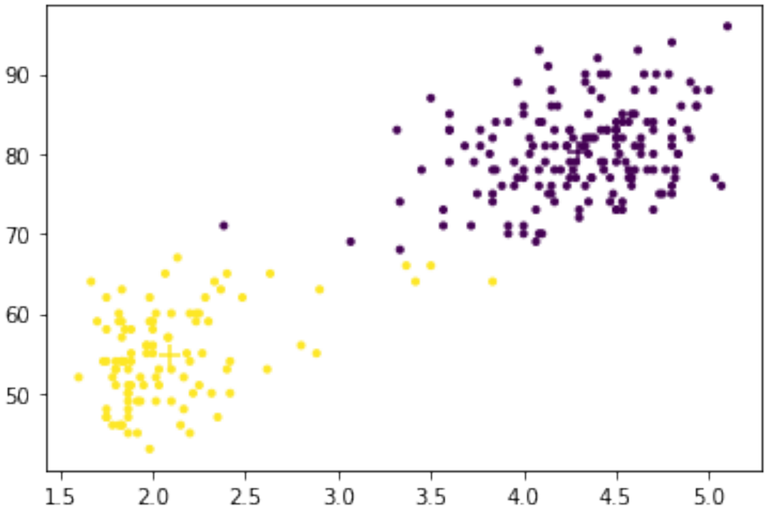

80
70
60
50
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

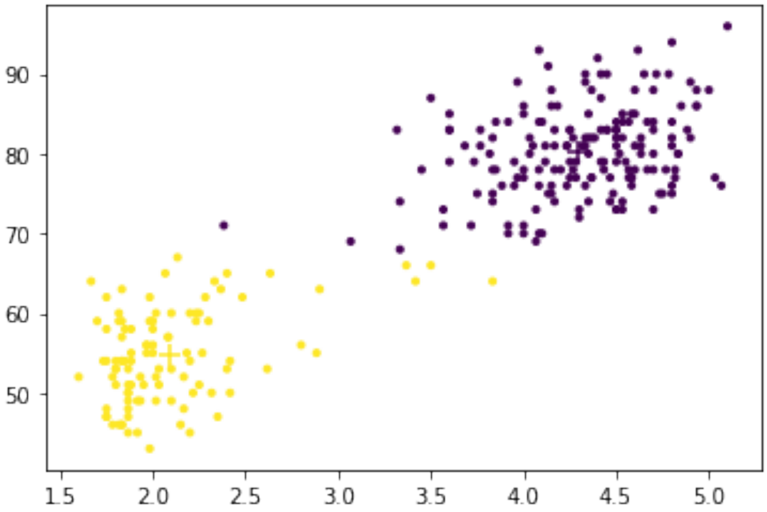

90
80
70
60
50
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


90
80
70
60
50
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


90
80
70
60







Mixtures of Gaussians

It is often the case that clusters have some overlaps and assignment is probabilistic. *Mixtures of Gaussians* is a probabilistic extention of K-means clustering.

Gaussian mixture distribution has a form

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \Sigma_k)$$

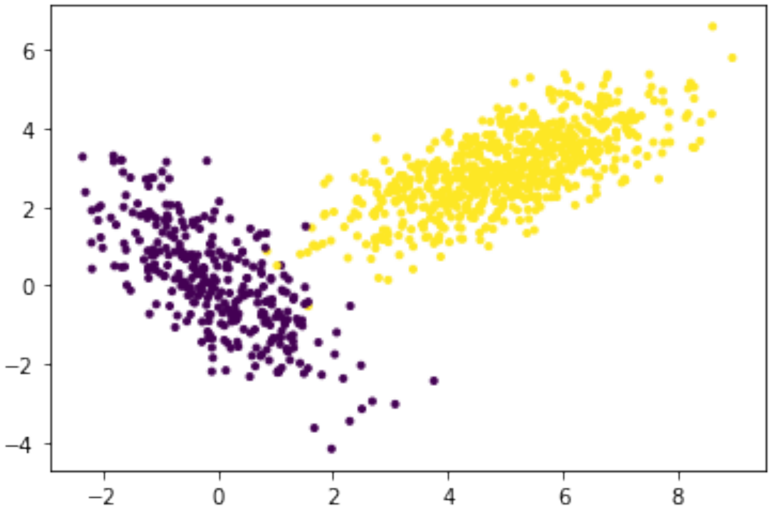
where $\boldsymbol{\mu}_k$ and Σ_k are the mean and the variance of k-th Gaussian and π_k is the mixture probability.

In [32]:

```
# sample from a Gaussian mixture distribution
def gaussmix(pi, mu, sigma):
    K = len(pi)
    z = np.random.multinomial(1, pi)  # binary stochastic variable
    k = list(z).index(1)     # the index of z_k=1
    x = np.random.multivariate_normal(mu[k], sigma[k])
    return x, k
```

In [33]:

```
pi = [0.3, 0.7]    # mixture probability
mu = [[0,0], [5,3]]      # means
sigma = [[[1,-1],[-1,2]], [[2,1],[1,1]]]    # variances
N = 1000
X = np.zeros((N,2))
Y = np.zeros(N, dtype=int)
for n in range(N):
    X[n,:], Y[n] = gaussmix(pi, mu, sigma)
plt.scatter(X[:,0], X[:,1], c=Y, marker='.');
```

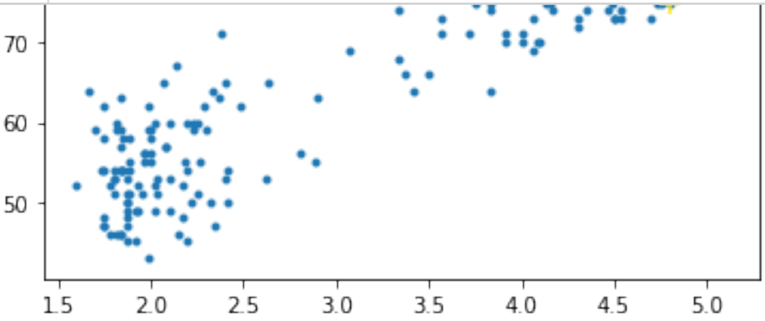


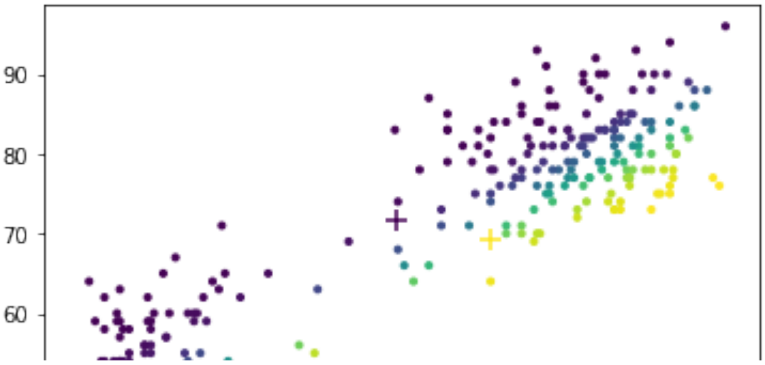
In [34]:

```
X = np.loadtxt('faithful.txt')
N, D = X.shape
```

In [35]:

```
# Initial means, covariance, responsibility
K = 2  # number of clusters
Pi = np.ones(K)/K  # cluster probability
Mu = X[np.random.randint(0, N, K),:]  # pick K points randomly
Sig = np.repeat(np.cov(X.T).reshape(1,D,D), K, axis=0)  # covariance for e
plt.scatter(X[:,0], X[:,1], marker='.')
#plt.hold(True)
plt.scatter(Mu[:,0], Mu[:,1], c=range(K), marker='+', s=100)
plt.show()
R = np.zeros((N,K))  # responsibility matrix
pr = np.zeros(K)  # data probability for each cluster
Lambda = np.zeros((K,D,D))  # inverse covariance
detSig = np.zeros(K)     # sqrt(det(Sig))
for t in range(15):
    # Expectation step
    for k in range(K):
        Lambda[k] = np.linalg.inv(Sig[k])   # inverse covariance
        detSig[k] = np.sqrt(np.linalg.det(Sig[k]))
    for n in range(N):
        # check the distances
        for k in range(K):
            #dx = np.matrix(X[n,:] - Mu[k,:])  # deviation from mean
            dx = X[n,:] - Mu[k,:]   # deviation from mean
            pr[k] = Pi[k]*np.exp(-dx@Lambda[k]@dx.T/2)/detSig[k]
        # responsibility
        R[n,:] = pr/np.sum(pr)    # responsibility p(z)
    # show assignment
    plt.scatter(X[:,0], X[:,1], c=np.dot(R,np.arange(K)), marker='.')
    # Maximization step
    num = np.sum(R, axis=0);    # effective numbers for each class
    Pi = num/N    # class prior
    for k in range(K):
        Mu[k,:] = np.sum(R[:,k]*X.T, axis=1)/num[k]
        dX = X - Mu[k,:]
        Sig[k] = R[:,k]/num[k]*dX.T@dX  # cluster covariance
    # plot the new means
    #plt.hold(True)
    plt.scatter(Mu[:,0], Mu[:,1], c=range(K), marker='+', s=100)
    plt.show()
```





In [ ]:

Principal Component Analysis

Grasping the distribution of a high-dimensional data is not easy for human eyes. We often try to find a low-dimensional projection of the data that characteirizes the distribution.

Principal componet analysis (PCA) finds the directions of the data distribution with the largest variance.

Consider a projection of D-dimensional vector $\boldsymbol{x} = (x_1, \ldots, x_D)^T$ to M-dimensional vector $\boldsymbol{y} = (y_1, \ldots, y_M)^T$ by

$$\boldsymbol{y} = V\boldsymbol{x}$$

where $V = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_M)^T$, $||\boldsymbol{v}_m|| = 1$.

For a data set $X = (x_1, \ldots, x_N)^T$ with zero mean (mean subtracted), we try to find the projection by V that make the variance of $\boldsymbol{y}$ the largest.

Using the data covariance

$$C = \frac{1}{N} X^T X = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^T$$

the covariance of projection $\boldsymbol{z}$ is given as $V^T C V$.

After solving the eigenvalue problem $C\boldsymbol{v} = \lambda \boldsymbol{v}$, the covariance of projected data is maximized by $V = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_M)^T$ made of the eigenvectors with the largest eigenvalues $\lambda_1, \ldots, \lambda_M$.

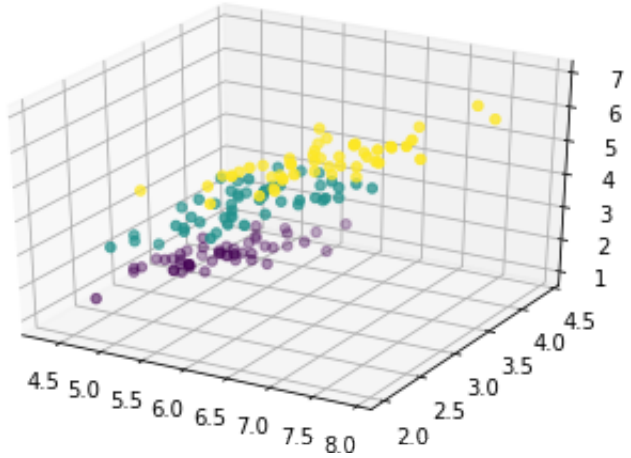
In [36]:

```python
# read CSV file
XT = np.loadtxt('iris.txt', delimiter=',')
X = XT[:, :-1]  # flower features
T = XT[:, -1]  # flower types
N, D = X.shape
print(N, D)
```

```
150 4
```

In [37]:

```
# plot the data in 3D
from mpl_toolkits.mplot3d import Axes3D # for 3D plotting
fig = plt.figure()
ax = fig.add_subplot(111,projection='3d')
ax.scatter(X[:,0], X[:,1], X[:,2], c=T);
```



In [38]:

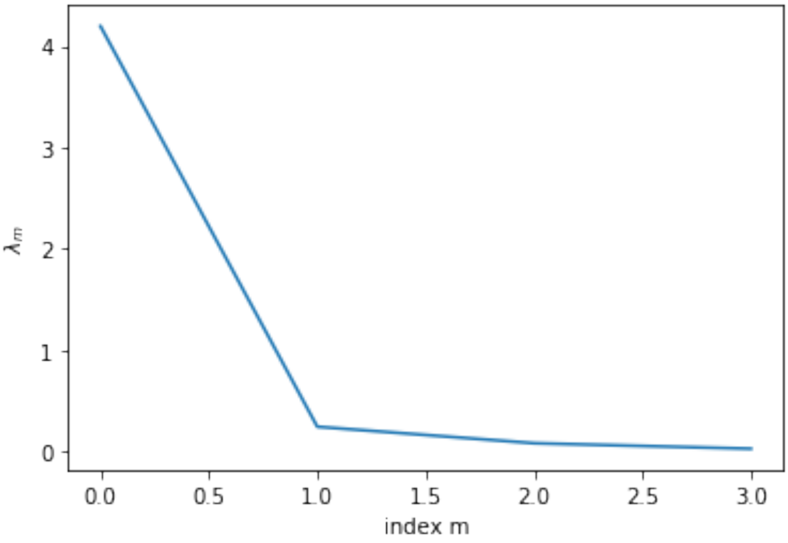
```
# Data covariance
X = X - np.mean(X, axis=0)  # subtract the mean
C = X.T@X/N
# eigenvalue L[i] and normal eigenvector V[:,i]
L, Vt = np.linalg.eigh( C)    # for symmetric matrix
#L, V = linalg.eig( C)
# in matrix form: C*V=V*L, i.e. C=V*L*V' from V'*V=I
print(L)    # eigenvalues
print(Vt)    # columns are eigenvectors
```

```
[0.02352514 0.07800042 0.24062861 4.19667516]
[[ 0.31725455  0.58099728  0.65653988 -0.36158968]
 [-0.32409435 -0.59641809  0.72971237  0.08226889]
 [-0.47971899 -0.07252408 -0.1757674  -0.85657211]
 [ 0.75112056 -0.54906091 -0.07470647 -0.35884393]]
```

In [39]:

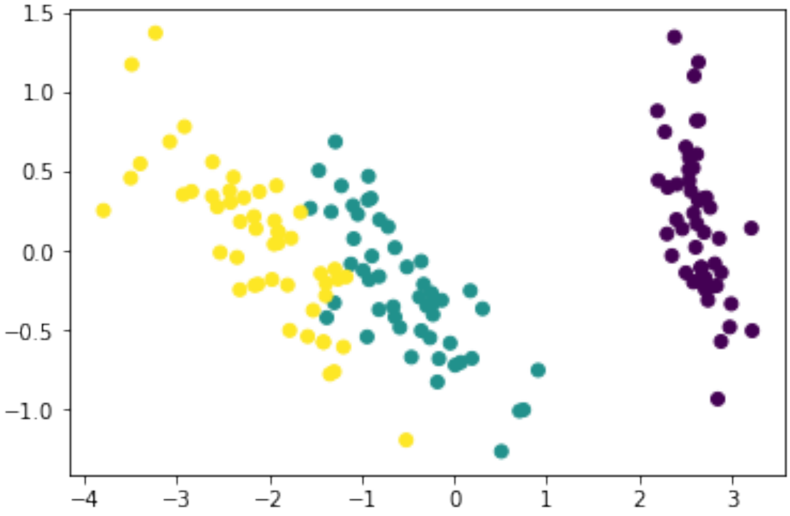
```python
ind = np.argsort(-L)  # largest first
L = L[ind]  # reorder
V = Vt.T[ind]
print(V)
plt.plot(L)
plt.xlabel("index m")
plt.ylabel("$\lambda_m$");
```

```
[[-0.36158968  0.08226889 -0.85657211 -0.35884393]
 [ 0.65653988  0.72971237 -0.1757674  -0.07470647]
 [ 0.58099728 -0.59641809 -0.07252408 -0.54906091]
 [ 0.31725455 -0.32409435 -0.47971899  0.75112056]]
```



In [40]:

```python
# Projection of data to the PC space
Z = X@V.T
# First two PC
plt.scatter(Z[:,0], Z[:,1], c=T);
```



PCA by a Neural Network

It has been shown that a simple linear neural network can perform computation similar to PCA (Sanger 1989).

Let us consider a two-layer network

$$\boldsymbol{y} = W\boldsymbol{x}$$

with input $\boldsymbol{x} = (x_1, \dots, x_D)^T$, output $\boldsymbol{y} = (y_1, \dots, y_M)^T$, and $M \times D$ connection weights W $(M < D)$.

Consider a *generalized Hebbian algorithm* In a componet form, it is

$$\begin{aligned}\Delta w_{ij} &= \alpha(y_i x_j - y_i \sum_{k \le i} w_{kj} y_k) \\ &= \alpha[y_i(x_j - \sum_{k<i} w_{kj} y_k) - y_i^2 w_{ij}]\end{aligned}$$

In a matrix form, it is represented as

$$\Delta W = \alpha(\boldsymbol{y}\boldsymbol{x}^T - LT[\boldsymbol{y}\boldsymbol{y}^T]W)$$

where $LT[\ ]$ takes the lower triangle (including the diagonal) of a matrix.

It has been shown that the rows of matrix W converges to the M eigenvectors with the largest eigen values of the covariance matrix $E[\boldsymbol{x}\boldsymbol{x}^T]$

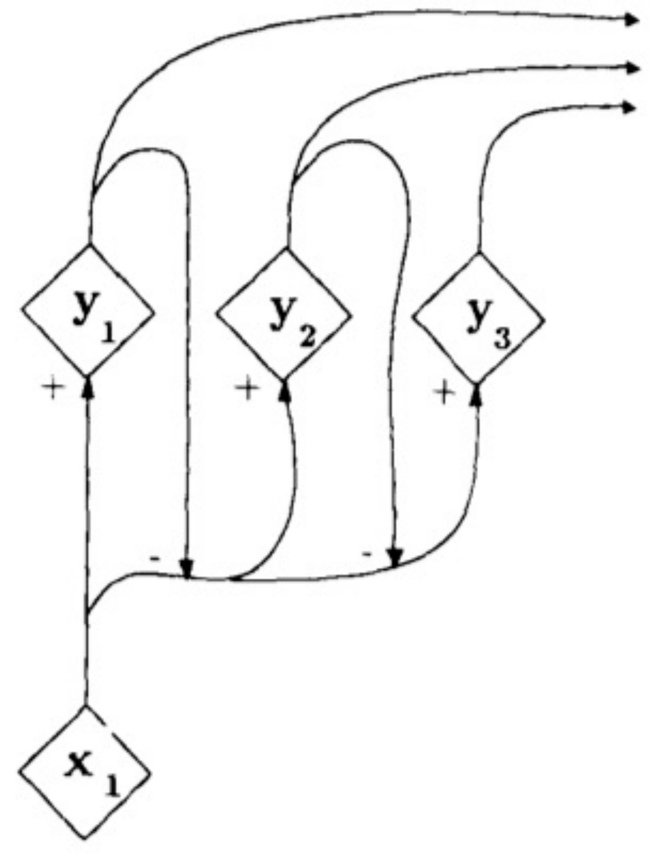


Figure: The ordered feedback inhibition in generalized Hebbian algorithm (from Sanger 1989).

In [41]:

```
def gha(X, W, alpha=0.01):
    """Generalized Hebbian Alogrithm by Sanger (1989)"""
    N, D = X.shape
    for n in range(N):
        y = W@X[n,:]
        W += alpha*(np.outer(y, X[n,:]) - np.tril(np.outer(y,y))@W)
    return W
```

In [42]:

```
# Iris example
M = 2
W = np.random.randn(M*D).reshape(M,D)
for k in range(10):
    W = gha(X, W, alpha=0.01)
    print(W)
```

```
[[ 0.39368973 -0.0136771   0.83395741  0.38868337]
 [ 0.0875187   0.22881663 -0.02144864  0.7534072 ]]
[[ 0.39282429 -0.01061874  0.83308401  0.39147158]
 [ 0.06914541  0.21361013 -0.02533481  0.56154467]]
[[ 0.39282327 -0.01061312  0.83308248  0.39147596]
 [ 0.06897411  0.22729074 -0.03095794  0.48048201]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.07884772  0.25516839 -0.03820455  0.43407221]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.09802751  0.29456942 -0.04734735  0.40230978]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.12734732  0.34508319 -0.0587314   0.37658723]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.16780984  0.40614872 -0.07256594  0.35171269]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.2194707   0.47553668 -0.08866958  0.32384003]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.28023286  0.5482209  -0.10620211  0.2902912 ]]
[[ 0.39282327 -0.01061311  0.83308247  0.39147597]
 [ 0.34521452  0.61658221 -0.1236217   0.25029986]]
```

In [43]:

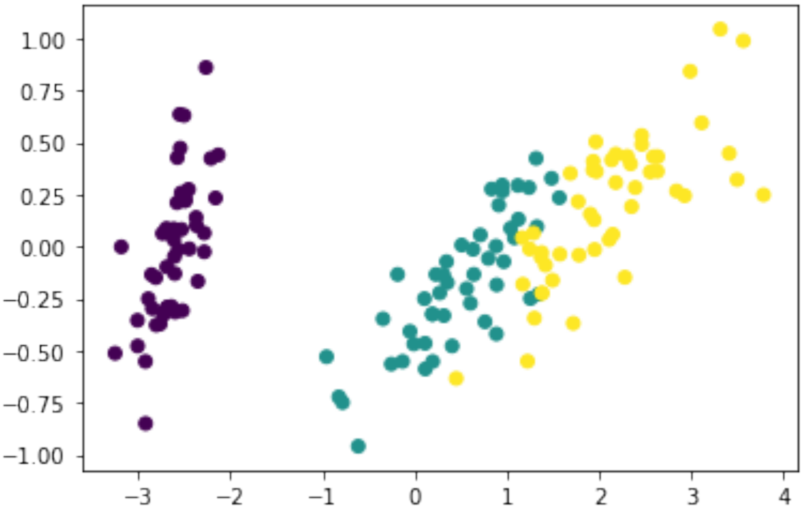
```
# Normalize
W/np.linalg.norm(W, axis=1, keepdims=True)
```

Out[43]:

```
array([[ 0.39248929, -0.01060408,  0.83237417,  0.39114313],
       [ 0.45435589,  0.81151789, -0.16270535,  0.32943346]])
```

In [44]:

```
# Projection of data to the PC space
Z = X@W.T
# First two PC
plt.scatter(Z[:,0], Z[:,1], c=T);
```



In [ ]:

```

```

In [ ]:

```

```