Exercise H8.1 : Vapnik - Chervonenkis dimension

$$\bar{C}_{(N+1,N)} = 2\sum_{k=0}^{N}\binom{N}{k}$$

$$= 2\sum_{k=0}^{N}\binom{N}{k}1^{k}1^{N-k}$$

$$= 2 \cdot 2^{N} = 2^{N+1}$$

$$\underbrace{\hat{C}_{(N+1,N)}}_{2^{N+1}} + \bar{C}_{(N+1,N-1)} = \bar{C}_{(N+2,N)}$$

$$\bar{C}_{(N+1,N-1)} = 2 \cdot \sum_{k=0}^{N-1}\binom{N}{k}$$

$$= 2 \cdot \Big(\underbrace{\sum_{k=0}^{N}\binom{N}{k}}_{2^{N}} - 1\Big)$$

$$= 2^{N+1} - 2$$

$$\bar{C}_{(N+2,N)} = 2^{N+1} + 2^{N+1} - 2 = 2^{N+2} - 2 < 2^{N+2}$$

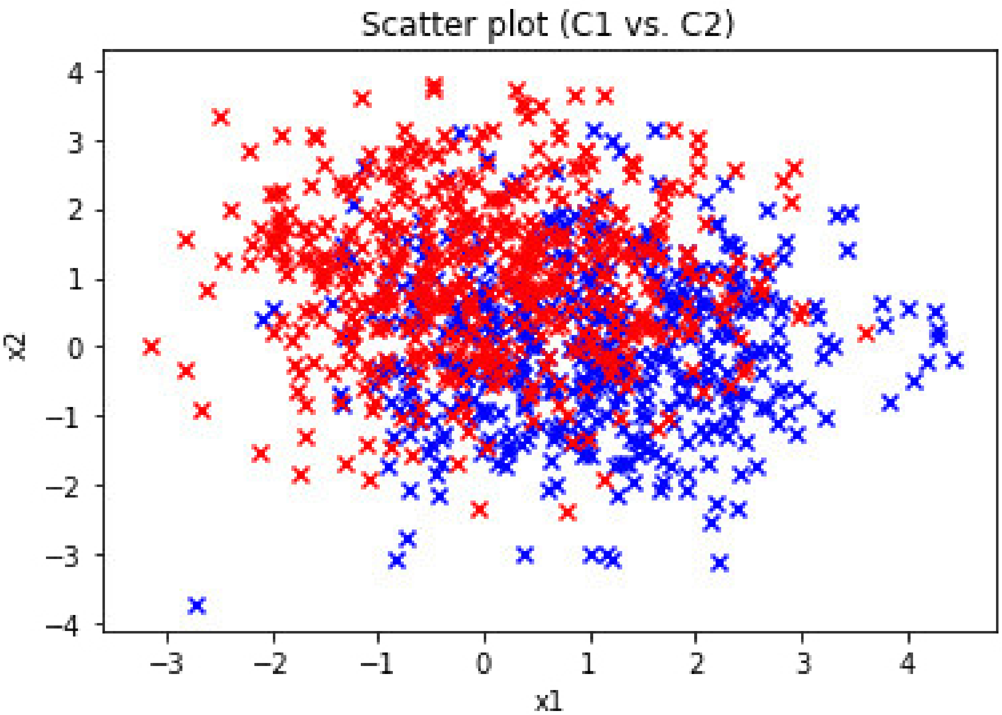
We have $\bar{C}_{(N+1,N)} = 2^{N+1}$ and $\bar{C}_{(N+2,N)} < 2^{N+2}$

so Vapnik - Chervonenkis dimension $d_{VC} = N+1$.

MI8_ex2_final

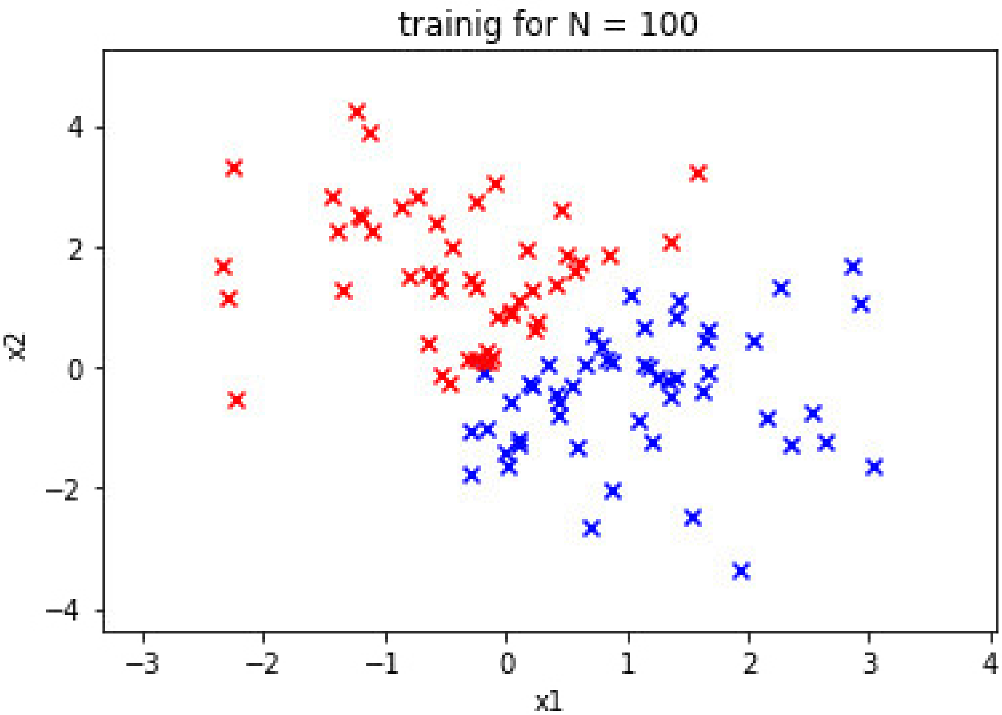
January 10, 2018

```
In [394]: import numpy as np
          import numpy.random
          import matplotlib.pyplot as plt
          import matplotlib.axes
          import numpy.linalg as la
          %matplotlib inline

In [395]: def generate_data(N) :
              if N%2 == 1 :
                  N += 1
              dataset = np.zeros((N, 3), dtype="float32")
              for i, obs in enumerate(dataset):
                  if i < N/2 :
                      obs[:2] = np.random.multivariate_normal([0.0, 1.0], [[2.0**0.5, 0.0], [0
                      obs[2] = 1.0
                  else :
                      obs[:2] = np.random.multivariate_normal([1.0, 0.0], [[2.0**0.5, 0.0], [0
                      obs[2] = -1.0
              return dataset

          def plot_data(dataset) :
              class1 = dataset[:, 2] == 1.0
              class0 = dataset[:, 2] == -1.0
              plt.scatter(dataset[class0][:, 0],dataset[class0][:, 1], color="blue", marker =
              plt.scatter(dataset[class1][:, 0],dataset[class1][:, 1], color="red", marker = '
              plt.xlabel('x1')
              plt.ylabel('x2')
              plt.title('Scatter plot (C1 vs. C2)')
              plt.show()

          plot_data(generate_data(1000))
```



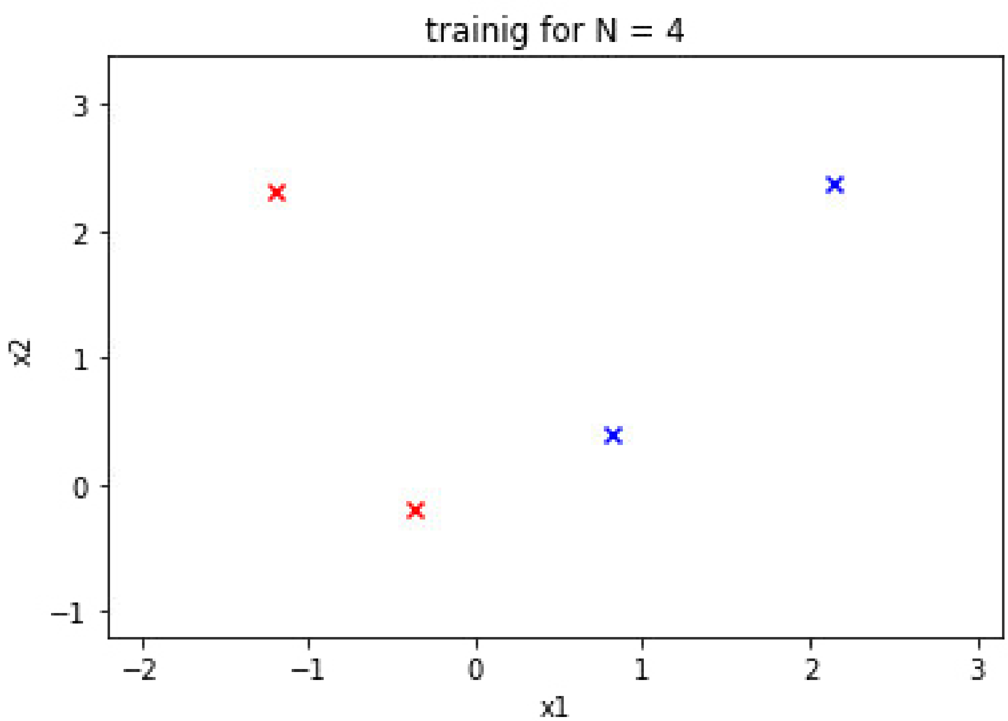

```
In [396]: def get_training_data(N) :
              training_inputs = np.zeros((N,3), dtype="float32")
              training_set = generate_data(N)
              for i in range(N) :
                  training_inputs[i] = [1, training_set[i][0], training_set[i][1]]
              labels = training_set[:, 2]
              return training_inputs, labels

          def out(x, w) :
              return np.dot(x.T, w)

          def getOutput(inputs, w) :
              output = []
              for i in range(len(inputs)) :
                  output.append(out(inputs[i],w))
              output = np.array(output, dtype = "float32")
              return output

          def compute_weights(input_vecs, labels) :
              X = input_vecs.T
              inv = la.inv(np.dot(X,X.T))
              prodX = np.dot(inv, X)
              return np.dot(prodX, labels)
```

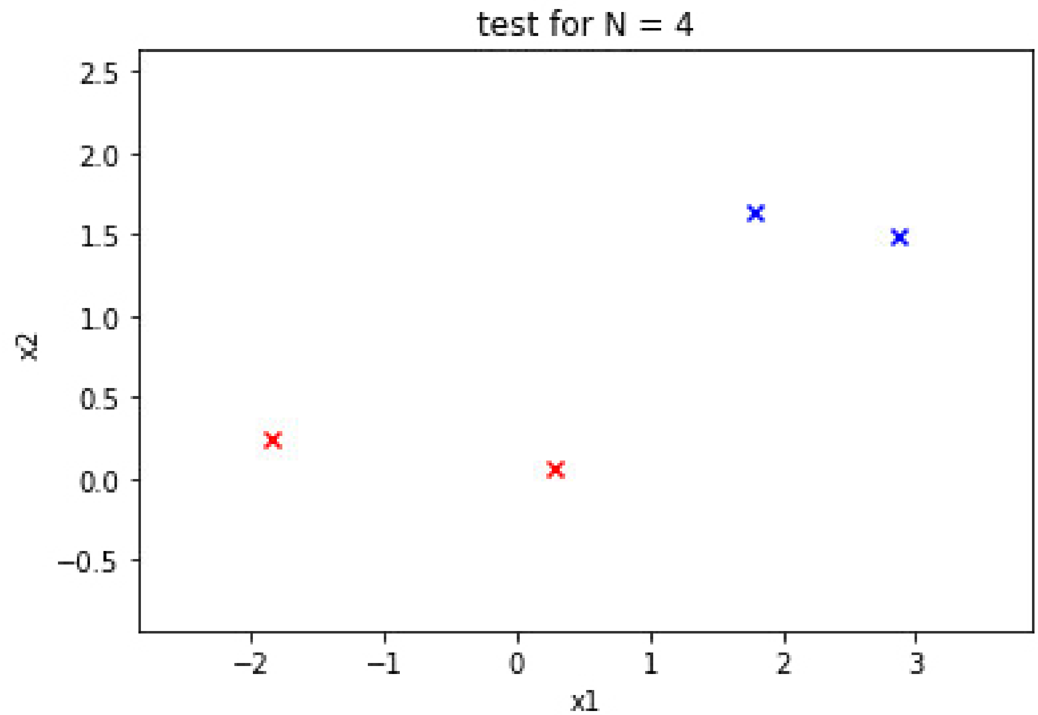
```
def getError(y_vec, labels) :
    return 1.0/len(labels) * sum([((y_vec[i]) - labels[i]) ** 2 for i in range(len(la

def train(input_vecs, labels) :
    weights = compute_weights(input_vecs, labels)
    y_vec = getOutput(input_vecs, weights)
    error = getError(y_vec, labels)
    return np.sign(y_vec), error, weights

train_data = get_training_data(100)
input_vecs = train_data[0]
labels = train_data[1]
res = train(input_vecs, labels)

def plot_classes(result, input_vecs, labels, N) :
    rclass1 = input_vecs[result==1]
    rclass0 = input_vecs[result==-1]
    plt.scatter(rclass0[:,1], rclass0[:,2], color = "blue", marker = 'x')
    plt.scatter(rclass1[:,1], rclass1[:,2], color = "red", marker = 'x')
    plt.axis([min(input_vecs[:,1]) - 1, max(input_vecs[:,1]) + 1, min(input_vecs[:,2
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('trainig for N = %i' %N)
    plt.show()

plot_classes(res[0], input_vecs, labels, 100)

def plot_test(result, input_vecs, N) :
    rclass1 = input_vecs[result==1]
    rclass0 = input_vecs[result==-1]
    plt.scatter(rclass0[:,1], rclass0[:,2], color = "blue", marker = 'x')
    plt.scatter(rclass1[:,1], rclass1[:,2], color = "red", marker = 'x')
    plt.axis([min(input_vecs[:,1]) - 1, max(input_vecs[:,1]) + 1, min(input_vecs[:,2
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('test for N = %i' %N)
    plt.show()
```

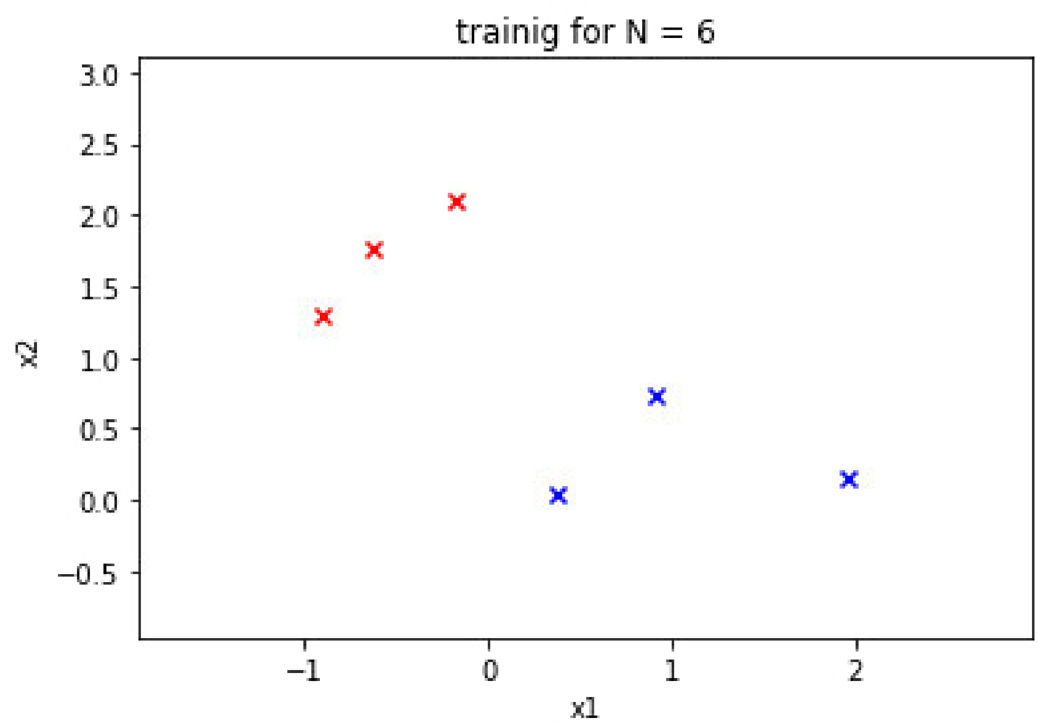


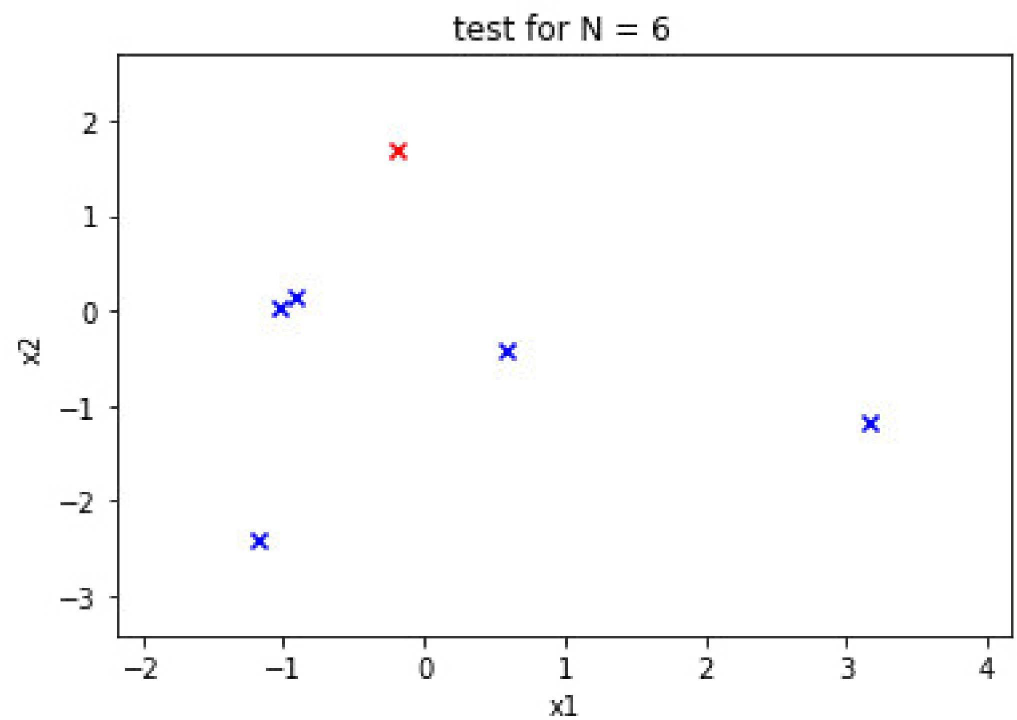

```
In [397]: def percentage(results, labels) :
              s = 0.0
              for i in range(len(labels)) :
                  if labels[i] == results[i] :
                      s += 1.0
              return s/len(labels)

          def calculate_accuracy(N) :
              train_data = get_training_data(N)
              input_vecs = train_data[0]
              labels = train_data[1]
              res = train(input_vecs, labels)
              results = res[0]
              weights = res[2]
              plot_classes(results, input_vecs, labels, N)
              accuracy_train = percentage(results, labels)
              test_data = get_training_data(N)
              test_vecs = test_data[0]
              test_labels = test_data[1]
              test_res = np.sign(getOutput(test_vecs, weights))
              accuracy_test = percentage(test_res, test_labels)
              plot_test(test_res, test_vecs, N)
              return accuracy_train, accuracy_test, results, test_res,weights
```

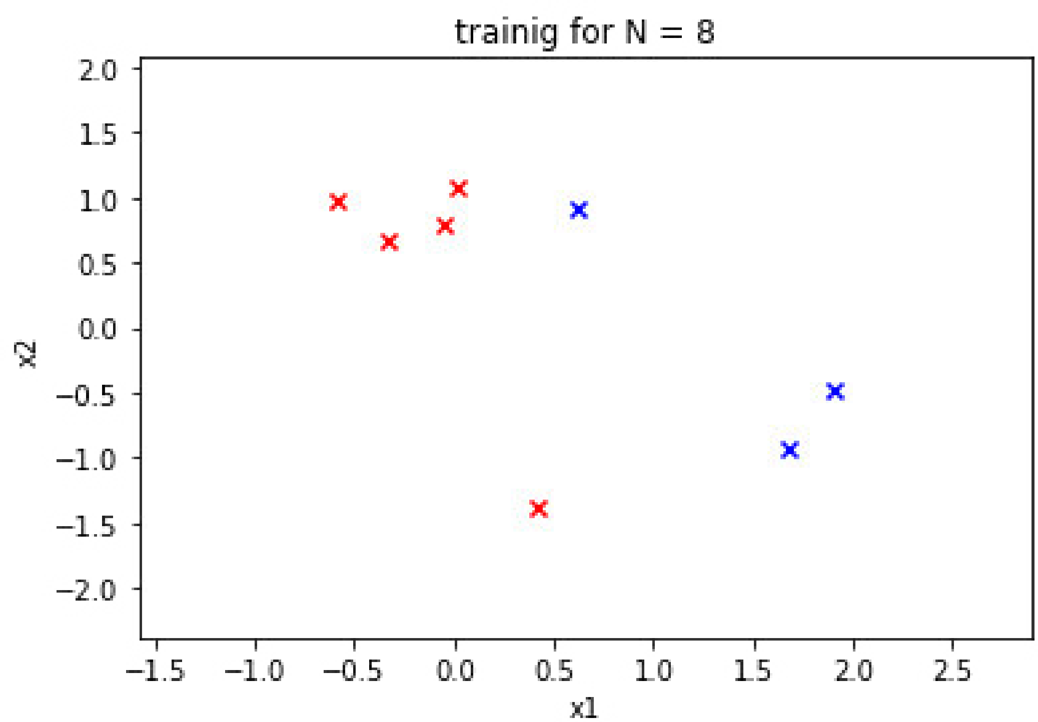
```
N_list = [4, 6, 8, 10, 20, 40, 100]

accuracy_train = []
accuracy_test = []
sd_train_list = []
sd_test_list = []
weight_list = []
for N in N_list :
    acc = calculate_accuracy(N)
    sd_train = 0
    sd_test = 0
    for i in range(N) :
        sd_train += (acc[0]-acc[2][i]) ** 2
        sd_test += (acc[1]-acc[3][i]) ** 2
    sd_train_list.append((1.0/N * sd_train) ** 0.5-1)
    sd_test_list.append((1.0/N * sd_test) ** 0.5-1)
    accuracy_train.append(acc[0])
    accuracy_test.append(acc[1])
    weight_list.append(acc[-1])
```

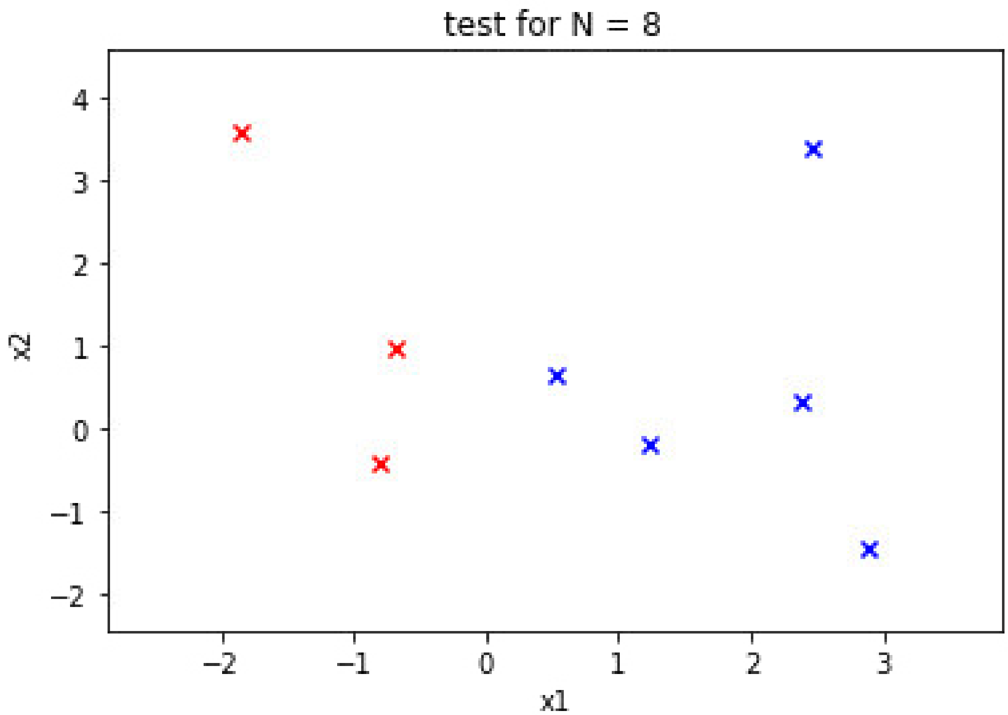


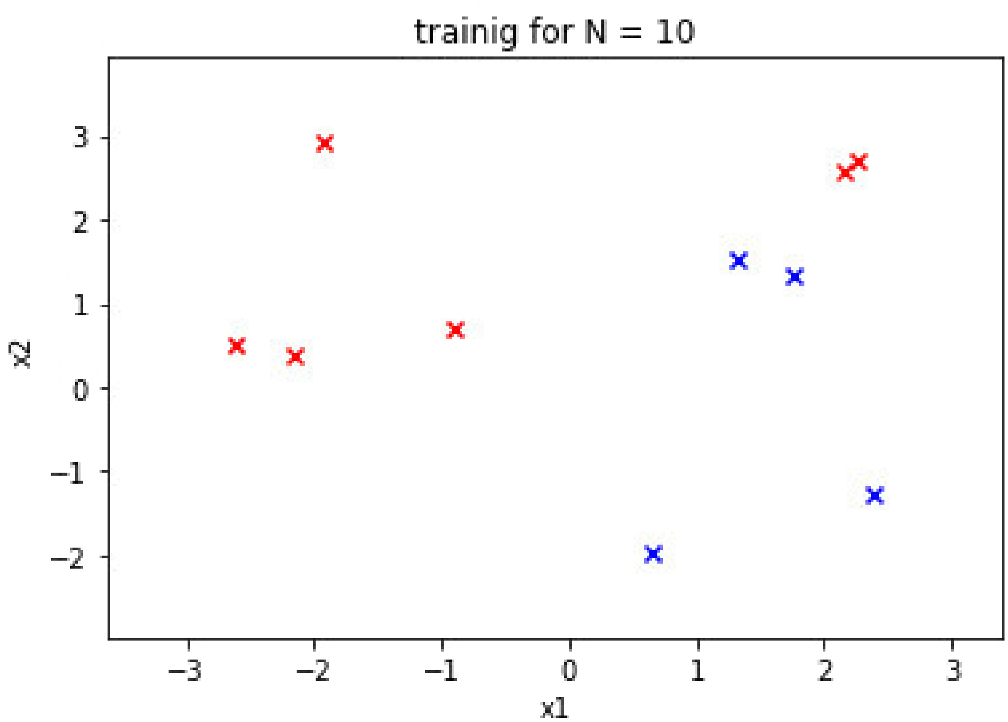

test for N = 4
x1
x2

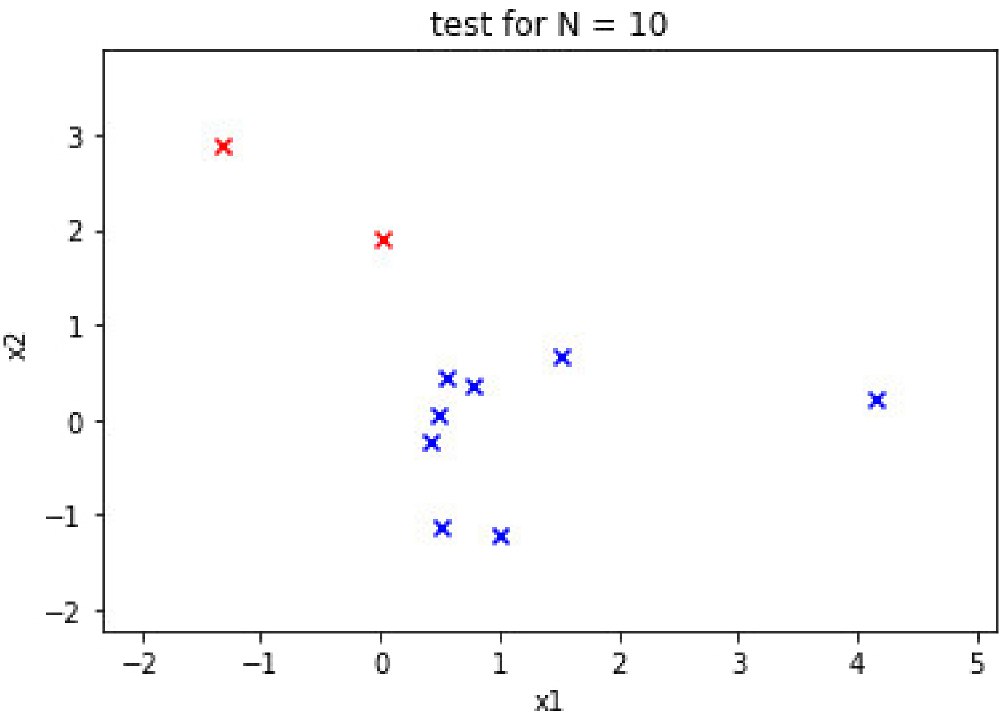

trainig for N = 6
x1
x2

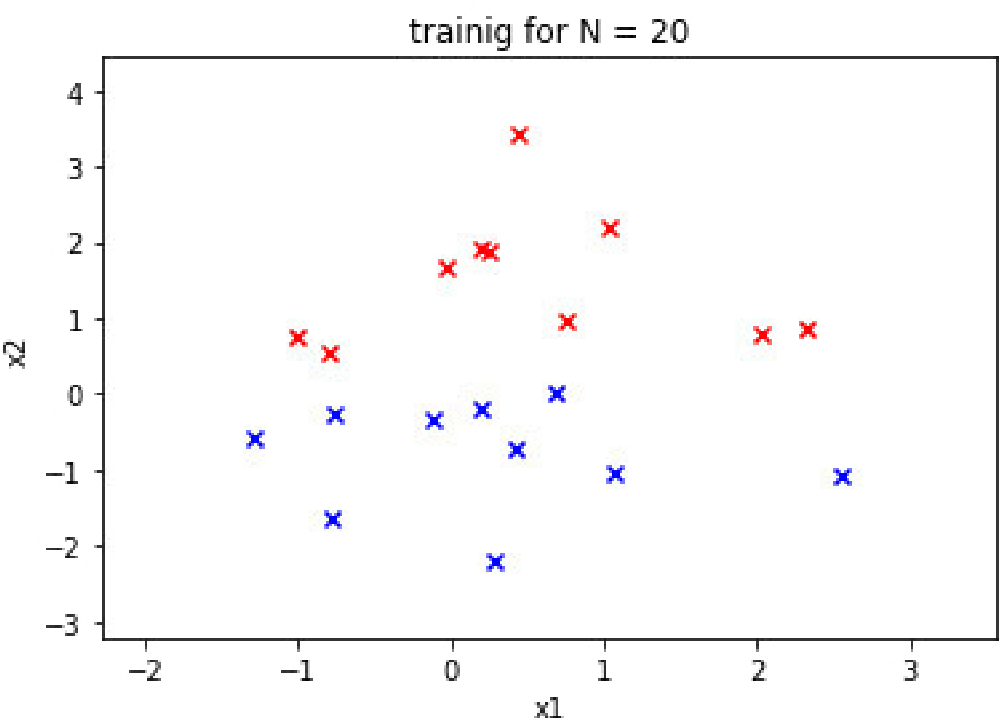

test for N = 6
x2
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4
x1

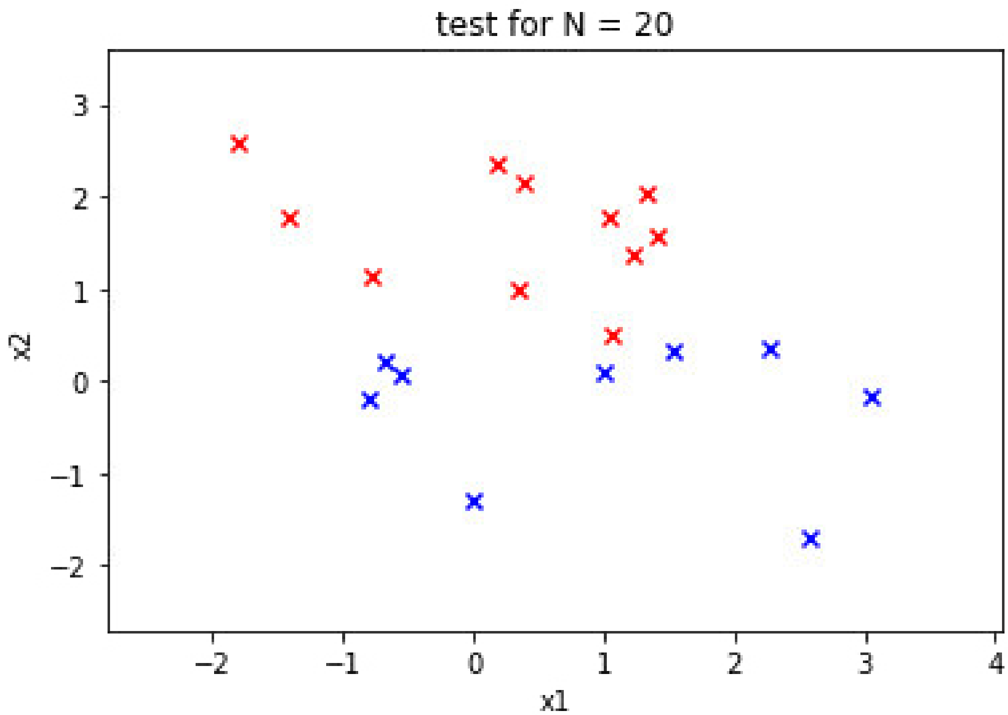

trainig for N = 8
x2
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x1

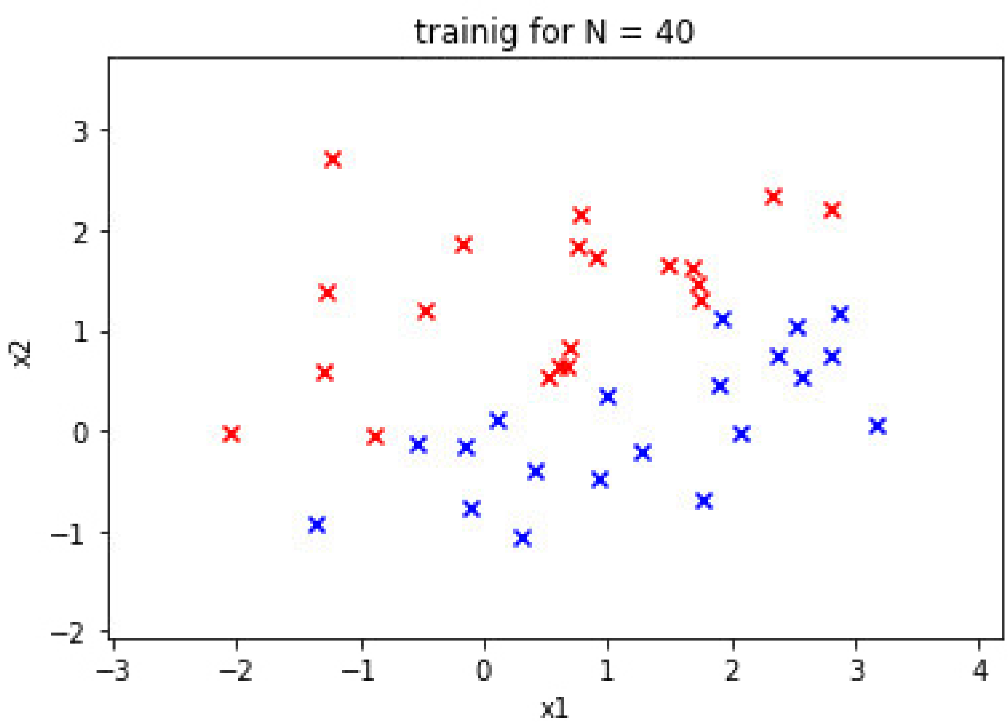

test for N = 8
x2
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3
x1

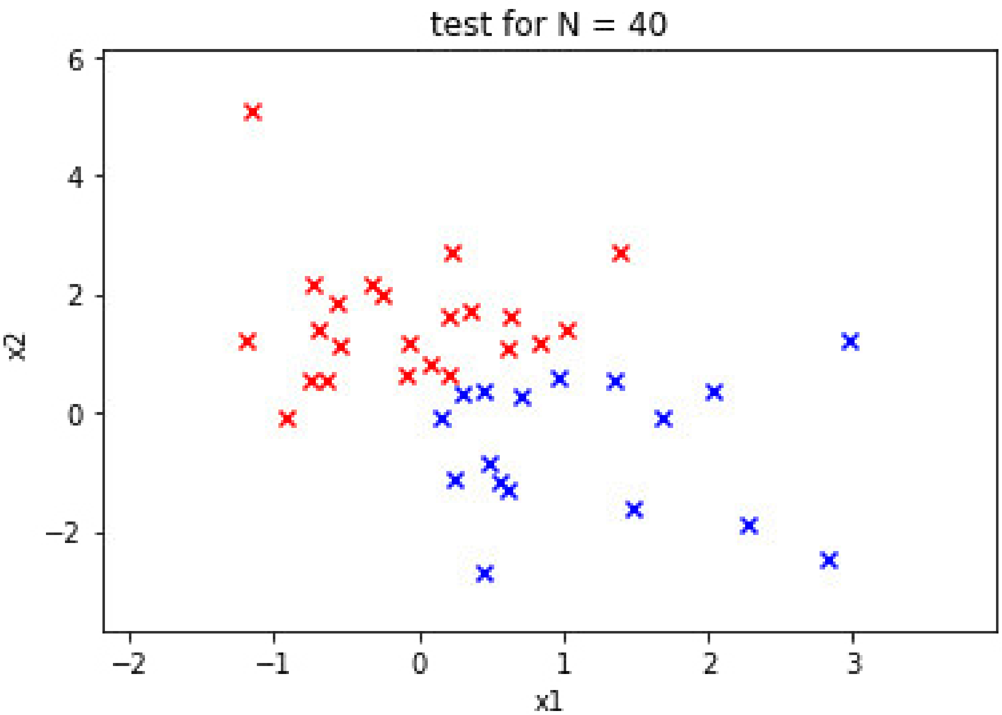

trainig for N = 10
x2
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
x1

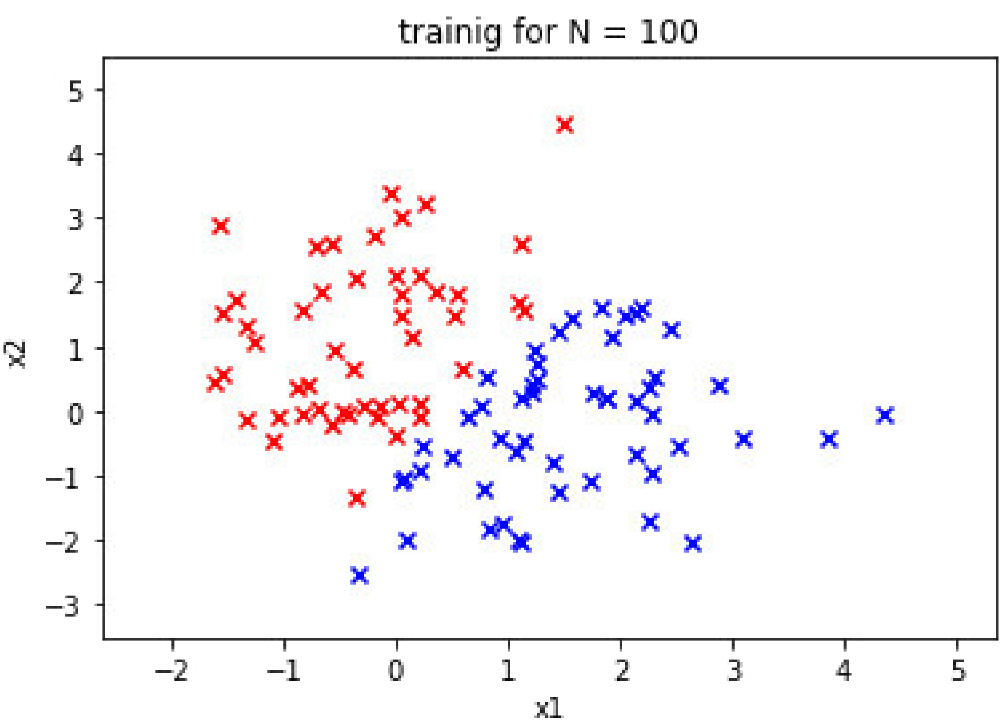

test for N = 10
x2
x1
3
2
1
0
−1
−2
−2
−1
0
1
2
3
4
5

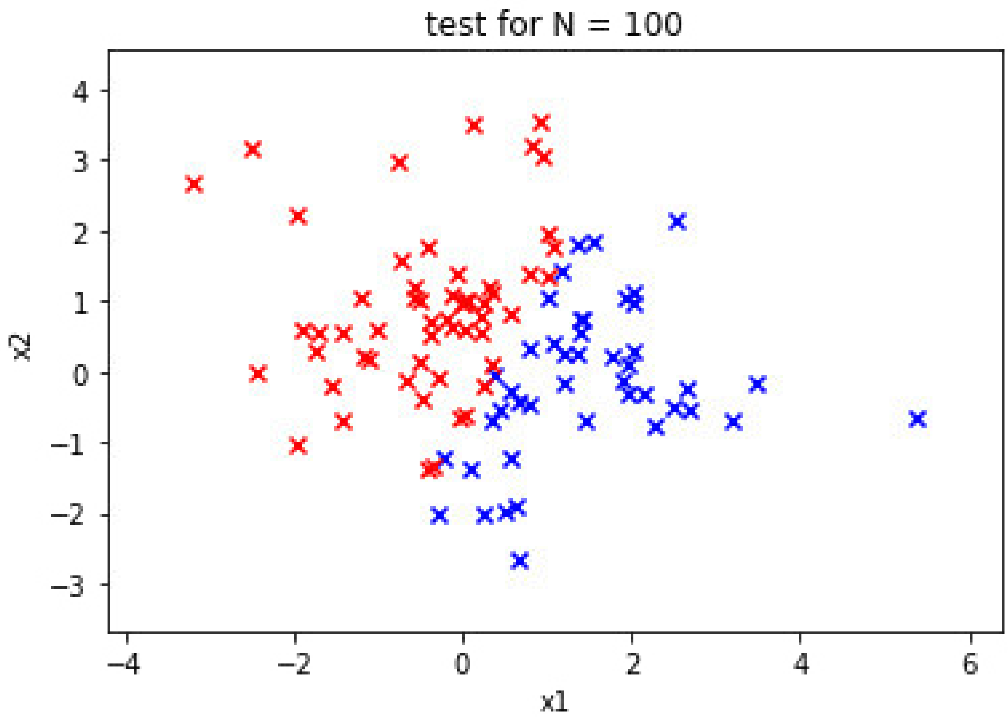

trainig for N = 20
x2
x1
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3


test for N = 20
x2
x1
3
2
1
0
−1
−2
−2
−1
0
1
2
3
4


trainig for N = 40
x2
x1
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4

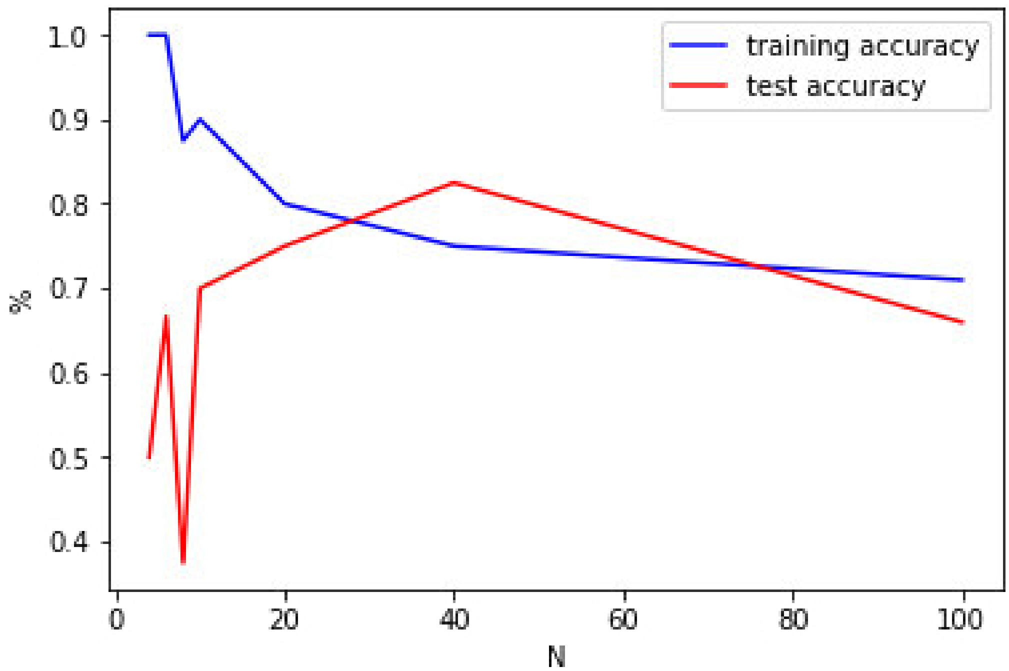

test for N = 40
x2
x1
6
4
2
0
−2
−2
−1
0
1
2
3


trainig for N = 100
x2
x1
5
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4
5



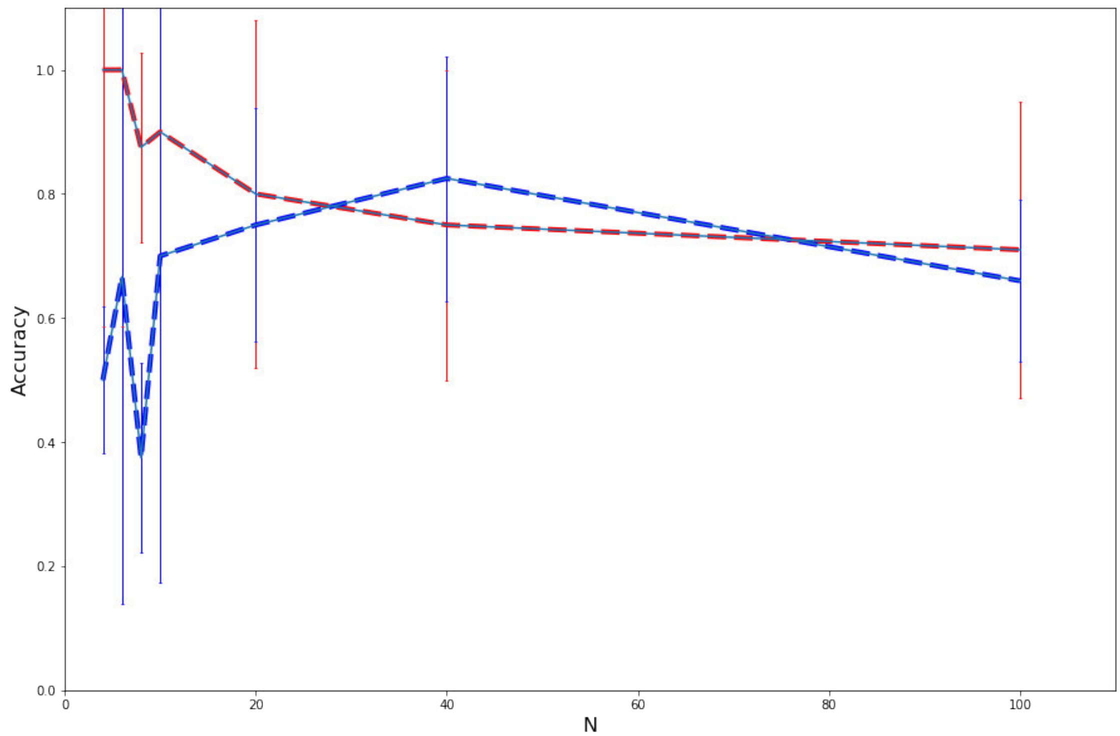
In [398]:
```
plt.plot(N_list, accuracy_train, color = 'blue', label = 'training accuracy')
plt.plot(N_list, accuracy_test, color = 'red', label = 'test accuracy')
#plt.axis([min(N_list),max(N_list),min(min(accuracy_train),min(accuracy_test)),max(m
plt.legend(loc = 'best')
plt.xlabel('N')
plt.ylabel('%')
plt.plot()
```

Out[398]: []



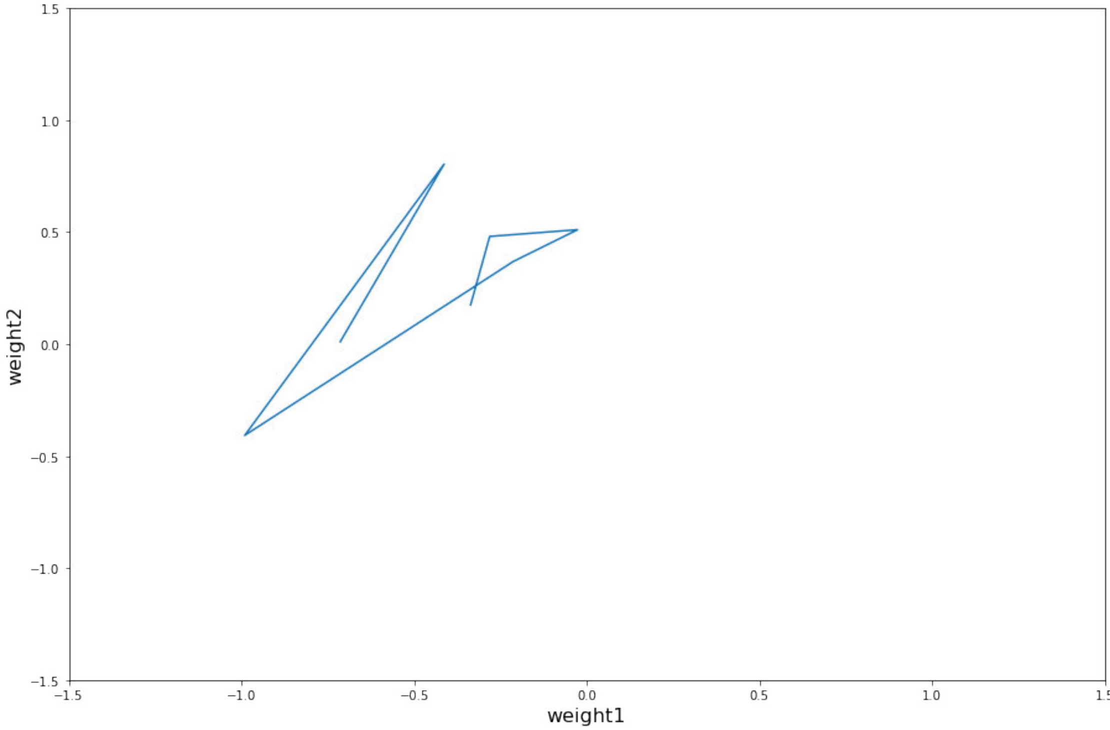

```
In [399]: plt.gcf().clear()
          #fig = plt.figure()
          fig = plt.figure(figsize=(15,10))
          ax = fig.add_subplot(111)
          ax.set_xlabel('N', fontsize = 16)
          ax.set_ylabel('Accuracy', fontsize = 16)

          sd_train_list
          ax.axis([0, 110, 0, 1.1])
          ax.plot(N_list, accuracy_train, 'r--', linewidth = 4)
          ax.errorbar(N_list, accuracy_train, yerr = sd_train_list, ecolor = 'r', elinewidth =
          ax.plot(N_list, accuracy_test, 'b--', linewidth = 4)
          ax.errorbar(N_list, accuracy_test, yerr = sd_test_list, ecolor = 'b', elinewidth = 1
          plt.show()
```

```
<matplotlib.figure.Figure at 0x1ebb8431940>
```



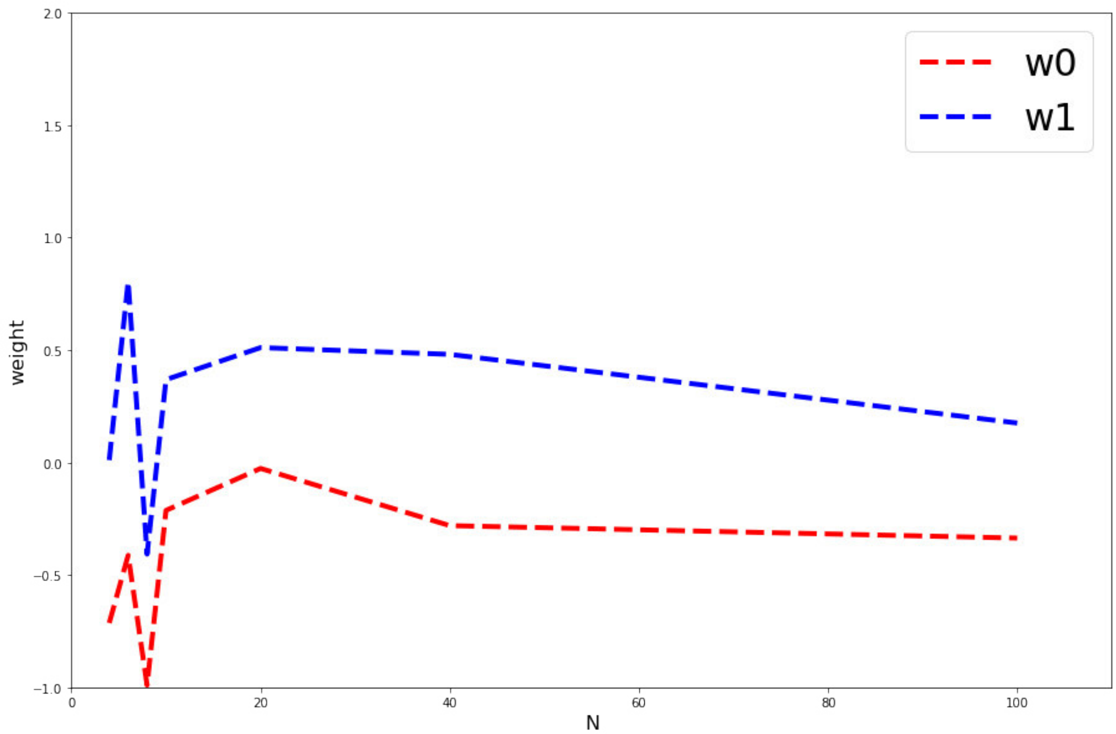

```
In [400]: import pandas as pd

          weight_df = pd.DataFrame(weight_list,columns = ['b','w1','w2'])

          plt.gcf().clear()

          fig = plt.figure(figsize=(15,10))
          ax = fig.add_subplot(111)
          ax.set_xlabel('weight1', fontsize = 16)
          ax.set_ylabel('weight2', fontsize = 16)

          sd_train_list
          ax.axis([-1.5, 1.5, -1.5, 1.5])
          ax.plot(weight_df.w1, weight_df.w2 )




          plt.show()
```

```
<matplotlib.figure.Figure at 0x1ebbb784ba8>
```




```
In [401]: plt.gcf().clear()
          #fig = plt.figure()
          fig = plt.figure(figsize=(15,10))
          ax = fig.add_subplot(111)
          ax.set_xlabel('N', fontsize = 16)
          ax.set_ylabel('weight', fontsize = 16)

          sd_train_list
          ax.axis([0, 110, -1, 2])
          ax.plot(N_list, weight_df.w1, 'r--', linewidth = 4,label='w0')
          ax.plot(N_list, weight_df.w2, 'b--', linewidth = 4,label='w1')
          plt.legend(fontsize=30)

          plt.show()
```

```
<matplotlib.figure.Figure at 0x1ebb9251d68>
```



With more samples, we can minimize the training error and we can predict closer to a minimum for the generalization error. With inifnite training samples, we approach certain optimal weights for EACH training set.

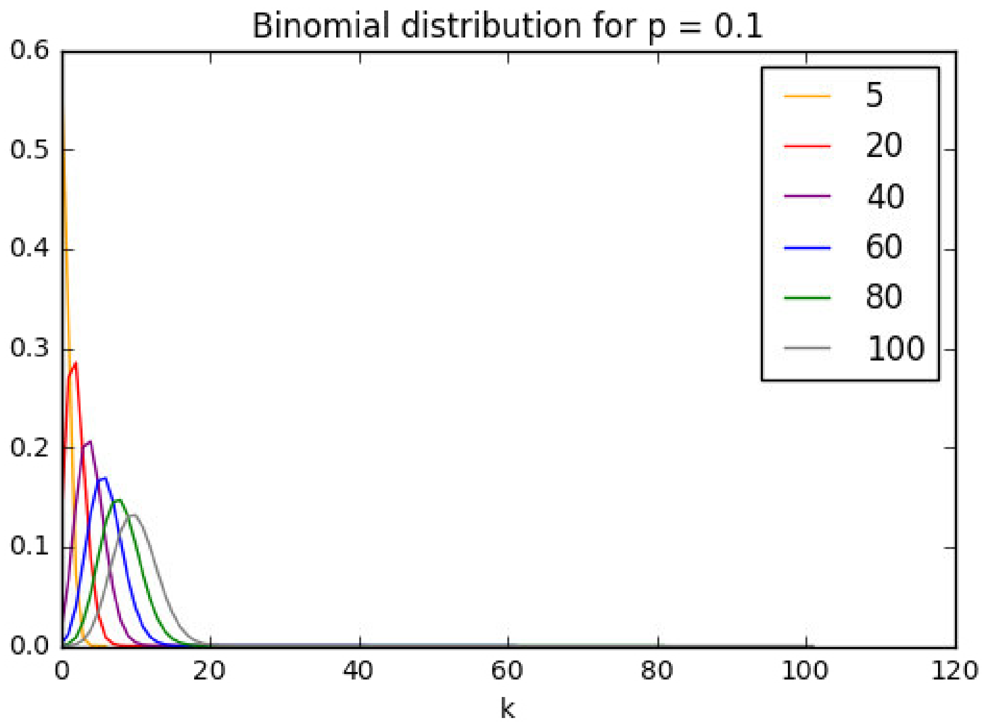
MI8_ex3

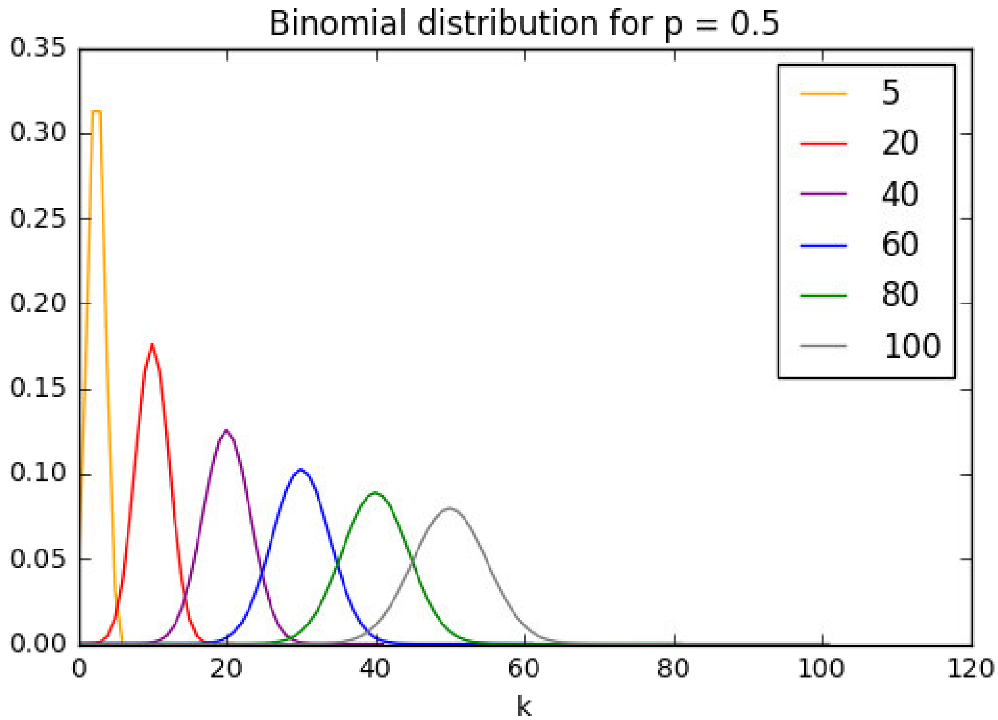
January 10, 2018

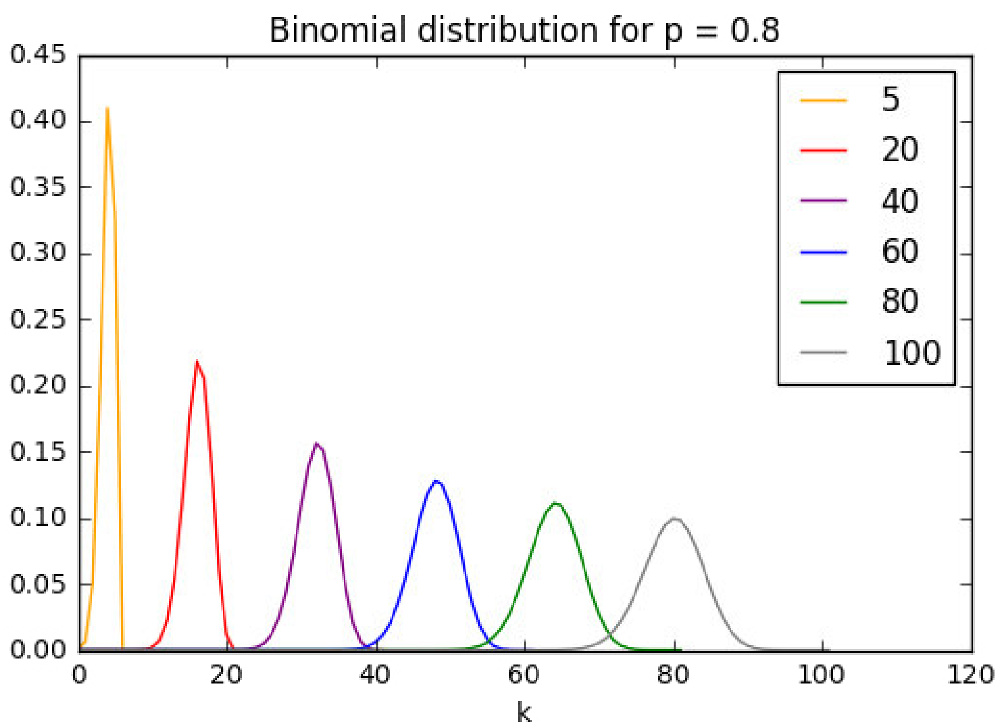
In [126]:
```
import numpy as np
import math
import matplotlib.pyplot as plt

def fact(n):
    if n == 1 or n == 0:
        return 1
    else :
        return n * fact(n-1)

cpn = np.zeros((200,200))
cpn[:,0] = 1
for i in range(1,200) :
    for j in range(1,i+1) :
        cpn[i][j] = cpn[i-1][j-1] + cpn[i-1][j]

def binomial(k,n,p):
    return cpn[n][k] * p**k * (1-p)**(n-k)

def normal(x,m,s):
    return np.exp(-(x-m)**2 / (2 * s**2)) / (s * (2*math.pi)**0.5)

def poisson(k,l):
    return l**k * np.exp(-l) / fact(k)
```

In [92]:
```
def binomial_fct(n,p) :
    res = []
    for k in range(n+2):
        res.append(binomial(k,n,p))
    return res

def plot_binomial(p) :
    colors = ['orange','red','purple','blue','green','grey']
    i = 0
    nlist = (5,20,40,60,80,100)
    for n in nlist :
        plt.plot(range(n+1),binomial_fct(n,p),color=colors[i])
        plt.xlabel('k')
```

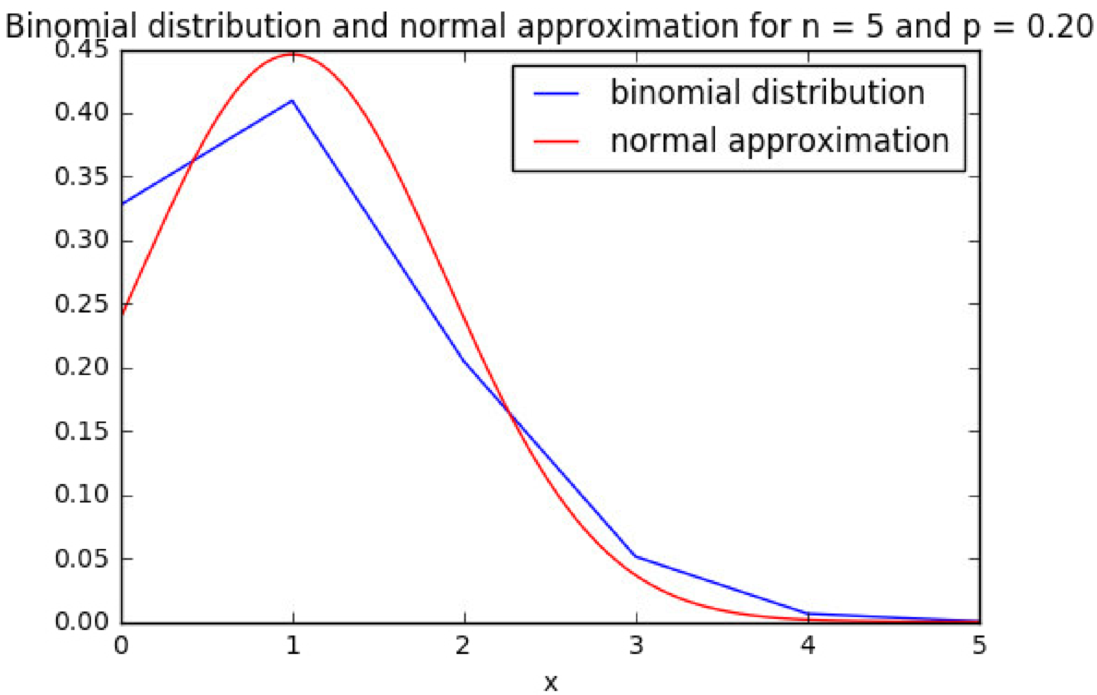
```
        plt.title('Binomial distribution for p = %.1f' %p)
        i += 1
    plt.legend(labels = nlist)
    plt.show()

## (b)

plot_binomial(0.1)
plot_binomial(0.5)
plot_binomial(0.8)
```

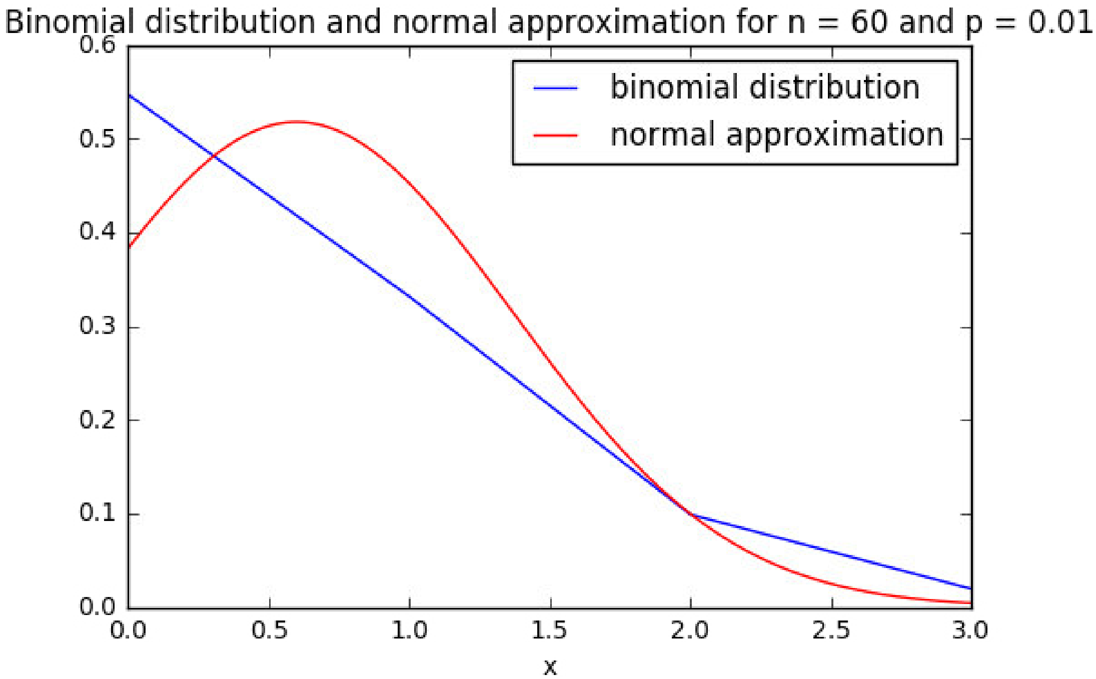


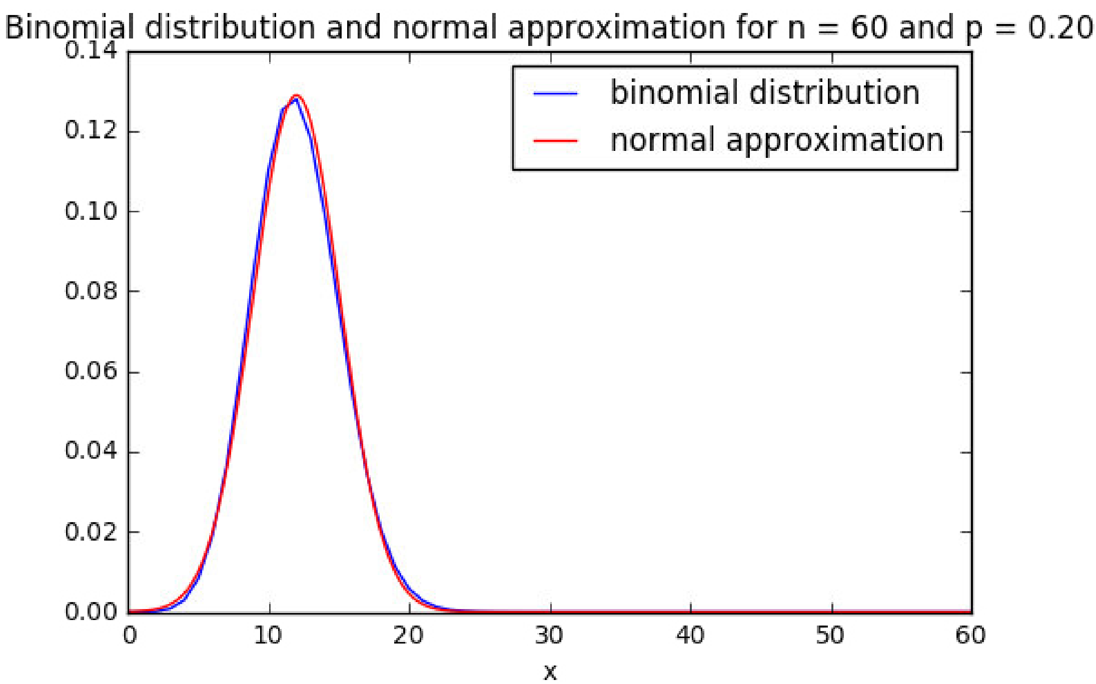

Binomial distribution for p = 0.5
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
120
k
5
20
40
60
80
100


Binomial distribution for p = 0.8
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
120
k
5
20
40
60
80
100

```
In [120]: def normal_approx(x,n,p):
              return normal(x,n*p,(n*p*(1-p))**0.5)

          def plot_binomial_normal(n,p) :
              x = np.linspace(0,n+2,100*(n+2)+1)
              plt.plot(range(n+2),binomial_fct(n,p),color='blue')
              plt.plot(x,normal_approx(x,n,p),color='red')
              plt.xlabel('x')
              plt.title('Binomial distribution and normal approximation for n = %i and p = %.2
              plt.xlim(0,5*n*p)
              plt.legend(labels = ('binomial distribution','normal approximation'))
              plt.show()

          ## (b)

          plot_binomial_normal(5,0.2)
          plot_binomial_normal(60,0.01)
          plot_binomial_normal(60,0.2)
```

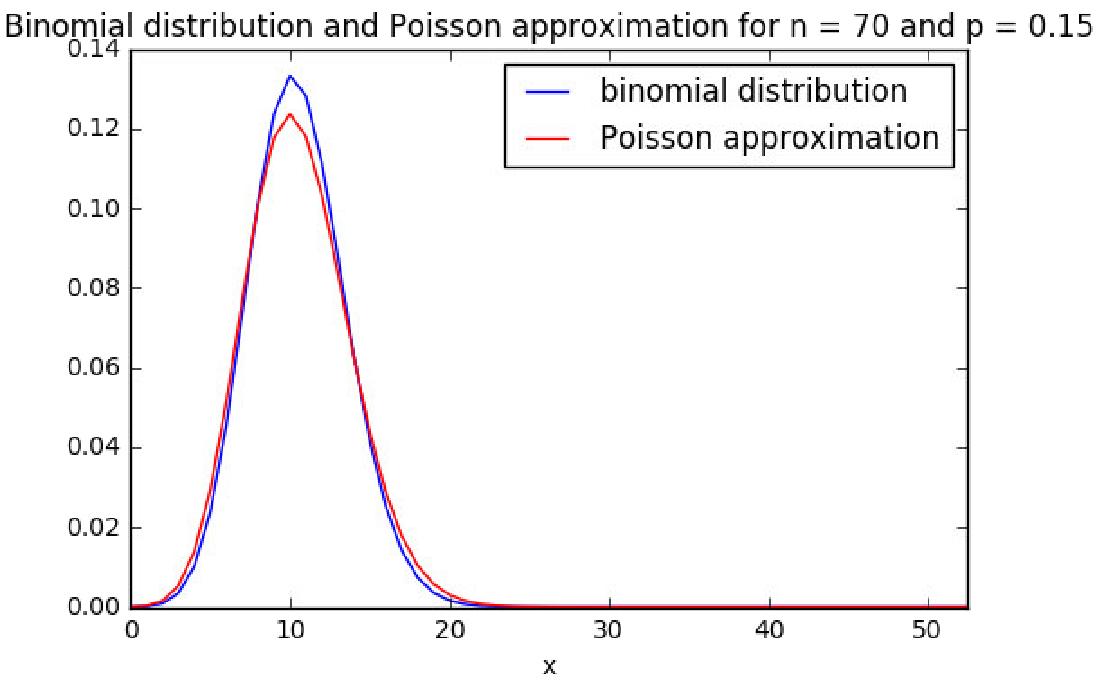


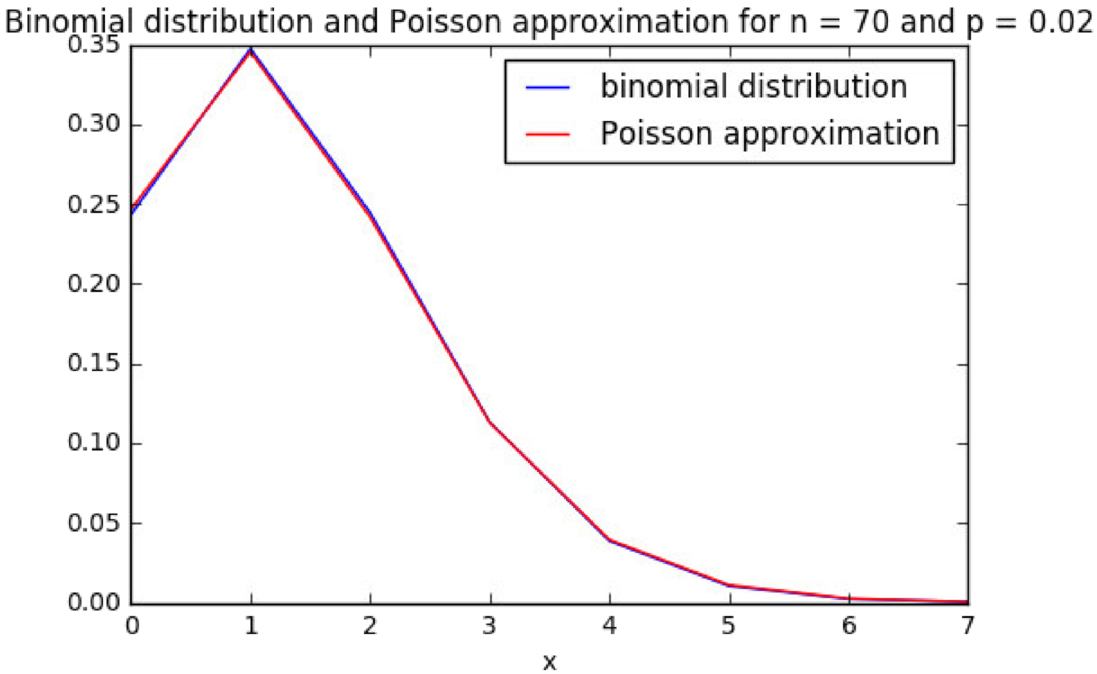




The binomial distribution is a good approximation of the binomial distribution if n is large enough (it seems good for n = 60). It is widely used because it is easier ot use and much faster to compute Above are 2 examples of bad normal approximation because of too small n or p.

```
In [134]: def poisson_approx(k,n,p):
              return poisson(k,n*p)

          def poisson_approx_fct(n,p):
              res = []
              for k in range(n+2):
                  res.append(poisson_approx(k,n,p))
              return res

          def plot_binomial_poisson(n,p) :
              plt.plot(range(n+2),binomial_fct(n,p),color='blue')
              plt.plot(range(n+2),poisson_approx_fct(n,p),color='red')
              plt.xlabel('x')
              plt.title('Binomial distribution and Poisson approximation for n = %i and p = %.:
              plt.xlim(0,5*n*p)
              plt.legend(labels = ('binomial distribution','Poisson approximation'))
              plt.show()

          ## (c)

          plot_binomial_poisson(70,0.15)
          plot_binomial_poisson(70,0.02)
```





The first graph shows a bad approximation because of a too high p and the second graph shows a good approximation with Poisson law. This approximation is good for a large number n (at least 20) and a small p (at most 0.05).