Question 1:
If we set S_w to the identity, the discriminant ends up as $w_0 = m_1 - m_2$, because the inverse of the identity matrix is also the identity. As long as the within classes scatter matrix is just a scaled version of the identity matrix, this shouldn't matter.

To test this method using both the conventional approach and setting the scatter matrix to the identity, a set of data points were generated within two classes. The data points were two dimensional, with equal prior probability for each class. For class 1, the first coordinate of each point was selected from a Normal distribution with mean 0.2 and sigma value 0.4. The second coordinate also used a Normal distribution, but had a mean equal to the value of the first and a sigma equal to 0.1. For class 2, the selection was identical except that the mean of the first point was 0.7. Therefore, the random variables used for the different points were not independent.

To test this, 5000 training points were generated, along with 5000 test points. The training points were used to generate a discriminant value, which could then be used to classify the test points. For the standard Fisher Linear Discriminant test, an accuracy of 84.3% was achieved classifying the points. While testing using the identity matrix instead of the scatter matrix, an accuracy of 73.9% was achieved. This indicates that the identity matrix appears to classify at a reduced accuracy. This is probably because, while the scatter matrix was symmetric, it was not a scaled version of the identity matrix.

Question 2:
The test data for this question included 10000 data points. Each point was two-dimensional with the distribution for each dimension being an independent Normal distribution. These data points were split into two classes, with equal probability for each class.

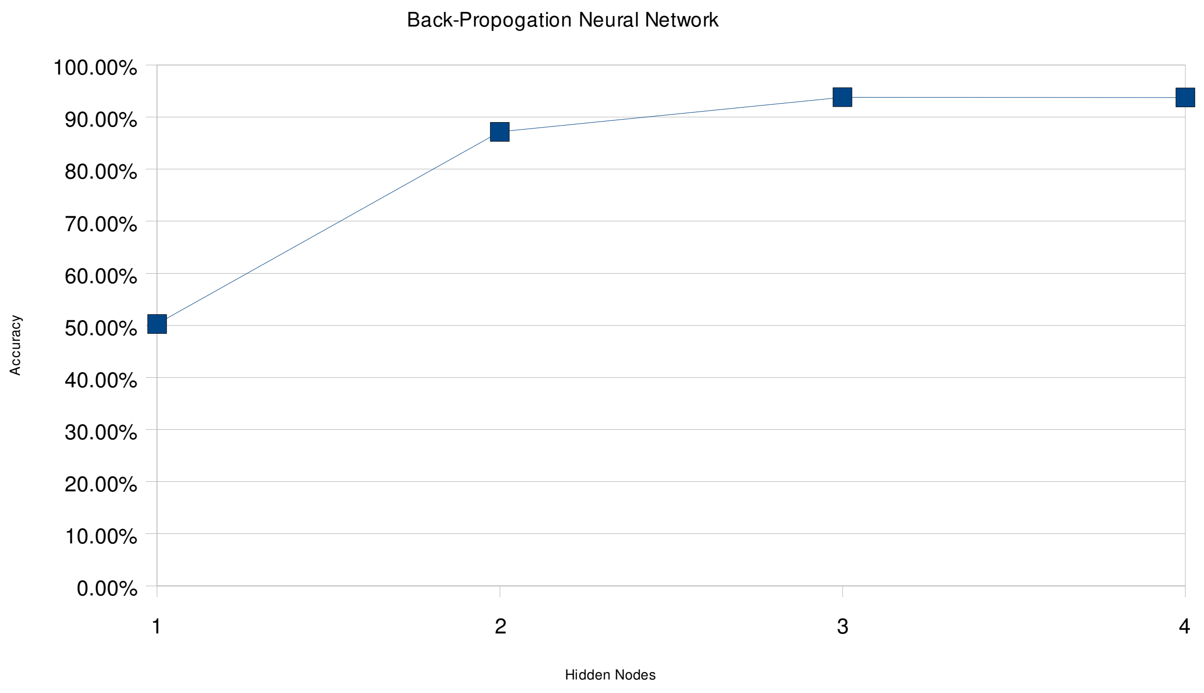
For the first dimension of each data point, Class 1 had a mean of 0.2 and a sigma value of 0.4 while Class 2 had a mean of 0.7 and a sigma value of 0.4. For the second dimension, Class 1 had a mean of 0.9 and a sigma value of 0.1, while Class 2 had a mean of 0.0 and a sigma value of 1.0. In all tests, 5000 data points were used for training and 5000 for evaluating the resulting classifier.

a)
A back-propagation neural network was used for this test. The code for the neural net itself is public domain code written by Neil Schemenauer. It can be found here at http://arctrix.com/nas/python/bpnn.py. This was used to generate and train the neural net. The algorithms implemented in the code are the same as those described in class. The primary deviation from a normal BPNN is that this code uses a tanh sigmoid function.

The neural net implemented used two input nodes (one for each dimension), a variable number of hidden layer nodes, and two output nodes. The first output node generated a value from 0 to 1 indicating the likelihood of a point being in Class 1, while the second indicated the likelihood of a point being in Class 2. The output node with the highest value determined the class assigned to the test point.

For training, a learning coefficient of 0.05 was used with a momentum coefficient of 0.01. The neural net weighting values were initialized to random values (uniform) between -2 and 2 prior to training. In all tests, the net was trained for 301 iterations. This was long enough to arrive at a local minimum in the error space.

The evaluation was then performed with a neural net with one, two, three, and four hidden layer nodes. The accuracy was plotted against the number of hidden layer nodes:



The neural net accuracy reached a peak of about 93.8% at three hidden layer nodes. There was no improvement when it was extended to four, indicating that this is the limit of the ability of the neural net to classify data. This illustrates the maximum useful number of hidden layer nodes.

b)
For the support vector machine, libsvm, produced by Chih-Chung Chang and Chih-Jen Lin, was used to provide an implementation of the support vector machine structure. Code for this may be found here: http://www.csie.ntu.edu.tw/~cjlin/libsvm/. This library is a fairly easy-to-use implementation of an SVM. It provides tools for generating SVM models based on training data and testing their accuracy against a set of test data.

For this test, the recommendations of the libsvm documentation (available here: http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf) were used to produce the basic parameters for the SVM. Primarly, this included selection of the kernel as:

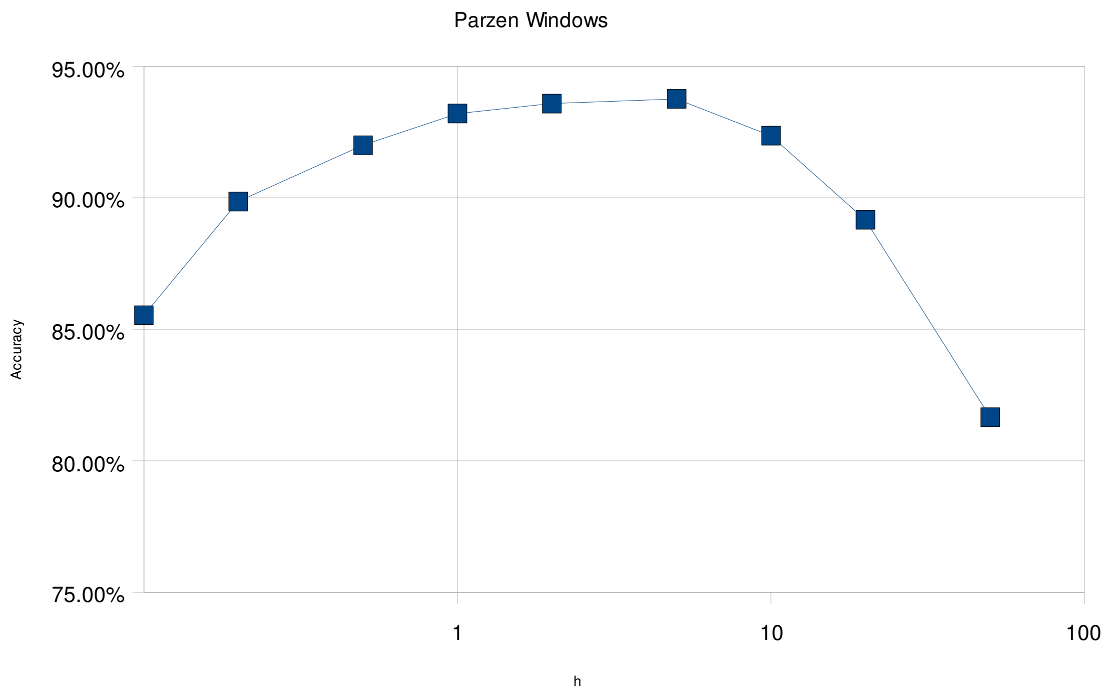
$$K(x,y)=e^{-y\|x-y\|^2}$$

The accuracy of the support vector machine was approximately 93.02%. This is consistent with the neural net.

Question 3:
For each classifier designed here, 5000 data points were used as 'training' data to create the PDFs and 5000 were used as test points to evaluate the resulting PDF. For each test point, the classifier determined whether the class 1 PDF or class 2 PDF has a higher value. The highest PDF is the class selected. The number of correct selections is then recorded and checked against the total number of test points (5000) to calculate the accuracy.

a)
The Parzen Window classifier was based on the discussion in-class and the example given on the Kiwi. Because the data were two-dimensional, instead one one, distance was used as the main weighting parameter. A Normal distribution was applied as the window to weight the sample points, with a mean of zero and a standard deviation of one.

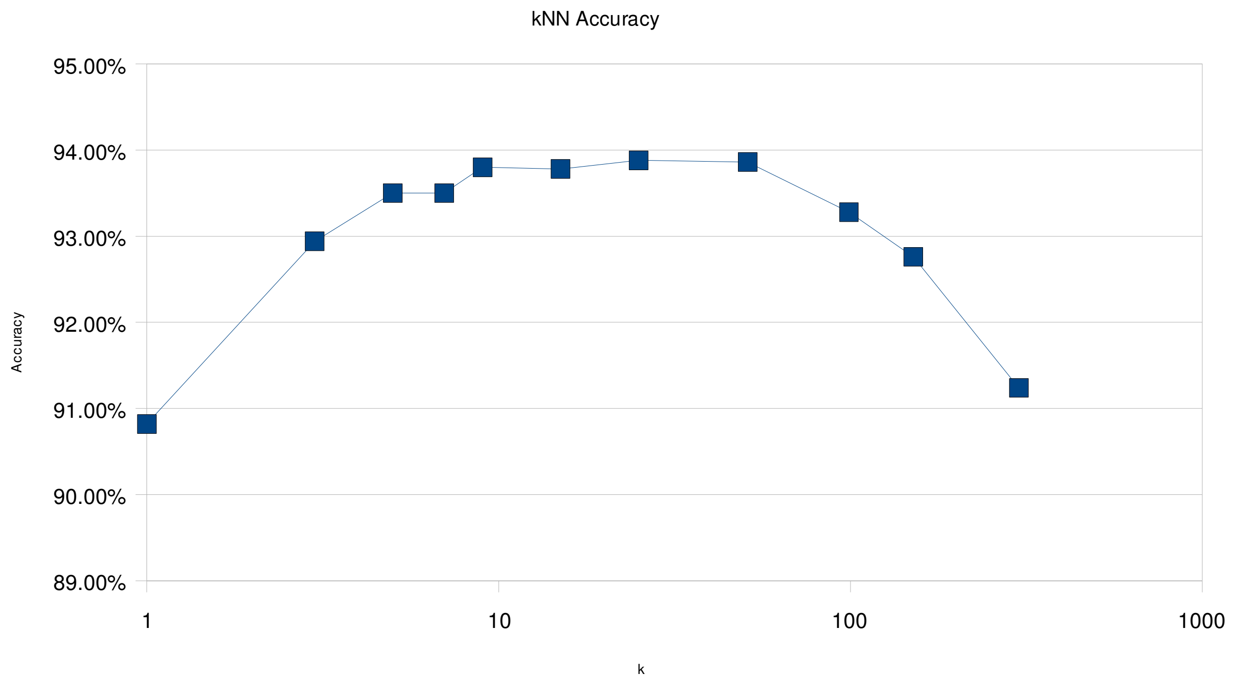
The h parameter was then varied between 0.1 and 50. Accuracy for the classifier was plotted according to the h parameter:



As can be seen, at too low an h value, far points are not weighted heavily enough and so usually only a single or a few points contribute. As a result, the PDF is not smooth enough. At too high an h value, the PDF becomes so smooth that the distinguishing characteristics of the underlying distribution are lost. The maximum accuracy here was about 93.6%, at an h value of 2.0.

b)
The k-nearest-neighbor classifier used a simple system where the nearest k data points were checked. The class with the most points was then determined as the class of the test point. The k values were always odd to allow one class or the other to always prevail. The values were selected between 1 and 301. Accuracy was then

plotted against the k value:



As can be seen here, a very similar pattern to that of the Parzen window classifier exists. For values of k too small, not enough nearby points are evaluated. At values of k too large, the PDF is smoothed out too much and so features of the underlying distribution are lost. Here, the highest accuracy was about 93.9%, at k=25. This is very close to the maximum accuracy of the Parzen Windows method.

c)
The nearest-neighbor method is a special case of the k-nearest-neighbors method where k=1. As seen in part b, this results in a relatively low accuracy, as the resulting PDF reflects too much on the particulars of the chose training points and not enough on the features of the distribution underlying those training points. The PDF thus consists only of 'spikes' near each point.

Code Listing 1: Data point Generator for Problems 2 and 3
generator.py

```
"""
Generate data points.
"""

from random import choice, gauss
from sys import argv, exit

# Distribution (mu, sigma)
CLASSES = (0, 1)
DIST = (
        [ # Class 1
            (0.2, 0.4),
            (0.9, 0.1),
        ],
        [ # Class 2
            (0.7, 0.4),
            (0.0, 1.0),
        ],
    )

# Check arguments
if len(argv) != 2:
    print 'USAGE: %s <num-points>' % argv[0]
    exit(2)
points = int(argv[1])

# Geneate data
for p in range(points):
    c = choice(CLASSES)
    point = ['%f' % gauss(mu, sigma) for mu, sigma in DIST[c]]
    print '%i:%s' % (c, ' '.join(point))
```

Code Listing 2: Data point generator for Problem 1

generator2.py

```
"""
Generate data points.

This is similar to the first generate script, except the variables are not
independent.
"""

from random import choice, gauss
from sys import argv, exit

# Distribution (mu, sigma)
CLASSES = (0, 1)
mu1 = [0.2, 0.7]
sigma1 = [0.4, 0.1]
sigma2 = [0.1, 0.7]

# Check arguments
if len(argv) != 2:
    print 'USAGE: %s <num-points>' % argv[0]
    exit(2)
points = int(argv[1])

# Geneate data
for p in range(points):
    c = choice(CLASSES)
    p1 = gauss(mu1[c], sigma1[c])
    p2 = gauss(p1, sigma2[c])
    point = ['%f' % i for i in [p1, p2]]
    print '%i:%s' % (c, ' '.join(point))
```

Code Listing 3: Fisher Linear Discriminant Classifier

p1_fld.py

```
"""
Use a Fisher Linear Discriminant to classify data.
"""

from numpy import matrix
from sys import argv, exit, stdin

# Check args
if len(argv) != 2:
    print 'USAGE: %s <num-test-points>' % argv[0]
    exit(2)
test_points = int(argv[1])

# Get all training points and test points
points = [l.strip().split(':') for l in stdin]
points = [(int(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Separate the training data from the test data
train = points[:-test_points]
test = points[-test_points:]
print 'Training points: %i' % len(train)
print 'Test points: %i' % len(test)
print

# Split training data into class 1 and 2
train1 = [p for c, p in train if c == 0]
train2 = [p for c, p in train if c == 1]

# Calculate means for each class
m1 = [sum([p[i] for p in train1]) / len(train1) for i in [0, 1]]
m2 = [sum([p[i] for p in train2]) / len(train2) for i in [0, 1]]
print 'm1: (%.2f, %.2f)' % tuple(m1)
print 'm2: (%.2f, %.2f)' % tuple(m2)
print

# Convert to column vectors
m1 = matrix(m1).transpose()
m2 = matrix(m2).transpose()
m = [m1, m2]

# Convert data to column vectors
train1 = matrix(train1).transpose()
train2 = matrix(train2).transpose()
train = [train1, train2]

# Generate the 'within classes scatter matrix'
Sw = matrix([[0.0, 0.0], [0.0, 0.0]])
for c in [0, 1]:
    for i in range(1):
        xi = matrix([[train[c][0, i]], [train[c][1, i]]])
        Sw += (xi - m[c]) * (xi - m[c]).transpose()
print 'Sw = '
print Sw

# Calculate inverse of Sw
SwI = Sw**(-1)
print 'Sw^-1 = '
print SwI
print

# Calculate w
w = SwI * (m1 - m2)
x_hat0 = (w.transpose() * (m1 + m2) / 2)[0, 0]
print 'w = ', w
print 'x hat 0 = ', x_hat0
print

# Classify all test data
print 'Testing...'
```

```
good = 0
for c, p in test:
    x = matrix([p]).transpose()
    x_hat = (w.transpose() * x)[0, 0]
    pc = 1 * (x_hat < x_hat0)
    if pc == c:
        good += 1
print 'Correctly classified points: %i' % good
print 'Incorrectly classified points: %i' % (len(test) - good)
```

Code Listing 4: Modified Fisher Linear Discriminant Classifier (using identity matrix instead of scatter matrix)

p1_modified.py

```
"""
Use a modified Fisher Linear Discriminant to classify data.
This does not use a scatter matrix.  Instead, an identity matrix is used.
"""

from numpy import matrix
from sys import argv, exit, stdin

# Check args
if len(argv) != 2:
    print 'USAGE: %s <num-test-points>' % argv[0]
    exit(2)
test_points = int(argv[1])

# Get all training points and test points
points = [l.strip().split(':') for l in stdin]
points = [(int(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Separate the training data from the test data
train = points[:-test_points]
test = points[-test_points:]
print 'Training points: %i' % len(train)
print 'Test points: %i' % len(test)
print

# Split training data into class 1 and 2
train1 = [p for c, p in train if c == 0]
train2 = [p for c, p in train if c == 1]

# Calculate means for each class
m1 = [sum([p[i] for p in train1]) / len(train1) for i in [0, 1]]
m2 = [sum([p[i] for p in train2]) / len(train2) for i in [0, 1]]
print 'm1: (%.2f, %.2f)' % tuple(m1)
print 'm2: (%.2f, %.2f)' % tuple(m2)
print

# Convert to column vectors
m1 = matrix(m1).transpose()
m2 = matrix(m2).transpose()
m = [m1, m2]

# Convert data to column vectors
train1 = matrix(train1).transpose()
train2 = matrix(train2).transpose()
train = [train1, train2]

# Use the identity matrix for Sw
Sw = matrix([[1, 0], [0, 1]])

# Calculate inverse of Sw
SwI = Sw**(-1)

# Calculate w
w = SwI * (m1 - m2)
x_hat0 = (w.transpose() * (m1 + m2) / 2)[0, 0]
print 'w = ', w
print 'x hat 0 = ', x_hat0
print

# Classify all test data
print 'Testing...'
good = 0
for c, p in test:
    x = matrix([p]).transpose()
    x_hat = (w.transpose() * x)[0, 0]
    pc = 1 * (x_hat < x_hat0)
    if pc == c:
        good += 1
```

```
print 'Correctly classified points: %i' % good
print 'Incorrectly classified points: %i' % (len(test) - good)
```

Code Listing 5: Conversion script to libsvm data files conv_data.py

```
"""
Convert data to a format usable by LIBSVM.
"""

from sys import argv, exit, stdin

# Get all points
points = [l.strip().split(':') for l in stdin]
points = [(int(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Write all points back
for c, p in points:
    print '%i %s' % (c,
        ' '.join(['%i:%f' % (i+1, p[i]) for i in range(len(p))]))
```

Code Listing 6: Neural Network Evaluator

This uses the BPNN code from: http://arctrix.com/nas/python/bpnn.py

p2_nntest.py

```
"""
Test a neural net by training it and then test it against new data.  This
evaluates the net based on its output.  If the output is closest to zero, it
is determined as Class 1, otherwise it is class two.
"""

from sys import argv, exit, stdin

from bpnn import NN

# Check args
if len(argv) != 3:
    print 'USAGE: %s <num-test-points> <hidden-layer-nodes>' % argv[0]
    exit(2)
test_points = int(argv[1])
hidden_nodes = int(argv[2])

# Get all training points and test points
points = [l.strip().split(':') for l in stdin]
points = [(float(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Separate the training data from the test data
train = points[:-test_points]
test = points[-test_points:]
print 'Training points: %i' % len(train)
print 'Test points: %i' % len(test)

print 'Training...'
# Train a new neural net
n = NN(2, hidden_nodes, 2)
pat = [(p[1], (1-p[0],p[0])) for p in train]
n.train(pat, iterations=301, N=0.05, M=0.01)

print 'Testing...'
# Evaluate the net
good = 0
for c, p in test:
    o1, o2 = n.update(p)
    if o1 > o2:
        if c < 0.1:
            good += 1
    else:
        if c > 0.9:
            good += 1
print 'Correctly classified points: %i' % good
print 'Incorrectly classified points: %i' % (len(test) – good)
```

Code Listing 7: Parzen Window Classifier
p3_parzen.py

```
"""
Parzen window classifier.
"""

from math import e, pi
from sys import argv, exit, stdin

def dist(p1, p2):
    return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

# Check args
if len(argv) != 3:
    print 'USAGE: %s <num-test-points> <h>' % argv[0]
    exit(2)
test_points = int(argv[1])
h = float(argv[2])

# Get all training points and test points
points = [l.strip().split(':') for l in stdin]
points = [(int(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Separate the training data from the test data
train = points[:-test_points]
test = points[-test_points:]
print 'Training points: %i' % len(train)
print 'Test points: %i' % len(test)

# Calculate Parzen Window parameters
hn = h / len(train)**0.5
print 'Hn: %f' % hn

# Classify all test points based on the k-nearest training points
print 'Testing...'
good = 0
i = 0
c1 = (1/(2*pi)**0.5)
for tc, tp in test:
    pdf = [0.0, 0.0]
    # Calculate PDF values for each class based of weighted samples
    for c, p in train:
        d = dist(p, tp)
        pdf[c] += e**(-0.5*(d/hn)**2)/hn
    # NOTE: We'd usually normalize the 'PDFs' here, but since we're just
    #       comparing them we don't really need to do that.

    # Classify based on PDFs
    if pdf[tc] > pdf[1-tc]:
        good+=1

    # Show progress
    if i % 100 == 1:
        print i, '%.2f%%' % (float(good) / i * 100)
    i += 1
print 'Correctly classified points: %i' % good
print 'Incorrectly classified points: %i' % (len(test) - good)
```

Code Listing 8: k-nearest neighbor Classifier
p3_knn.py

```
"""
K-nearest neighbor classifier.
"""

from sys import argv, exit, stdin

def dist(p1, p2):
    return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

# Check args
if len(argv) != 3:
    print 'USAGE: %s <num-test-points> <k>' % argv[0]
    exit(2)
test_points = int(argv[1])
k = int(argv[2])

# Get all training points and test points
points = [l.strip().split(':') for l in stdin]
points = [(int(p[0]), [float(i) for i in p[1].split(' ')]) for p in points]

# Separate the training data from the test data
train = points[:-test_points]
test = points[-test_points:]
print 'Training points: %i' % len(train)
print 'Test points: %i' % len(test)

# Classify all test points based on the k-nearest training points
print 'Testing...'
good = 0
i = 0
for tc, tp in test:
    dists = [(dist(tp, p), c) for c, p in train]
    dists.sort()
    dists = dists[:k]
    if len([c for d, c in dists if c == 0]) > k/2:
        c = 0
    else:
        c = 1
    if tc == c:
        good += 1
    if i % 100 == 1:
        print i, '%.2f%%' % (float(good) / i * 100)
    i += 1
print 'Correctly classified points: %i' % good
print 'Incorrectly classified points: %i' % (len(test) - good)
```