Sheet 1

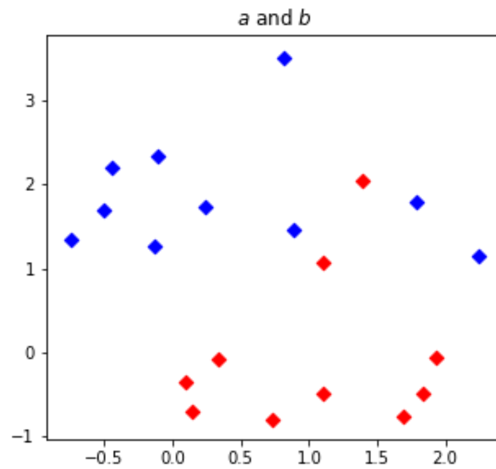
Initialisation

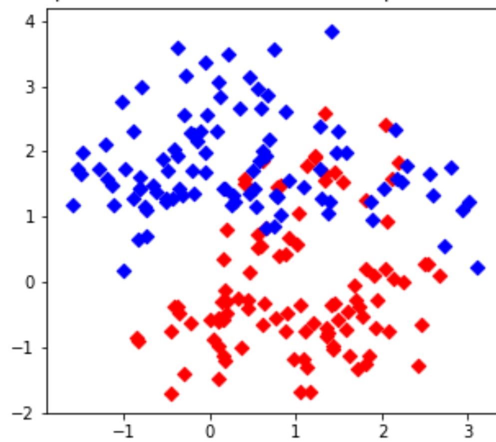
In [1]:
```
import numpy as np
import scipy.spatial as sp
import matplotlib.pyplot as plt
import pandas as pd
```

Task 1.1

Create $n = 200$ data points, make scatter plots of said data points, as required by task 1.1.

In [2]:
```
# number of datapoints
n = 200

# number of samples drawn from each distribution
num_samples = 10

# initialise mean and covariance arrays
mean0 = np.array([3/2, 0])
mean1 = np.array([0, 3/2])
cov = np.eye(2)

# initialise error, x, and y arrays
err = np.random.multivariate_normal(np.array([0,0]),cov/4,n)
x = np.zeros([n,2])
y = np.array(list(np.zeros([int(n/2),1])) + list(np.ones([int(n/2),1])))

# draw IID samples (a and b) from two bivariate normal distributions
a = np.random.multivariate_normal(mean0, cov, num_samples)
b = np.random.multivariate_normal(mean1,cov, num_samples)

# set up figures -- we have 2 figures stacked vertically
fig, axarr = plt.subplots(2,1, figsize=(5,10))

# plot a and b on a scatter plot
axarr[0].scatter(a[:,0],a[:,1],marker='D',c='red')
axarr[0].scatter(b[:,0],b[:,1],marker='D',c='blue')
axarr[0].set_title('$a$ and $b$')

# pick n equidistributed indices from 1,...,10
i = np.random.randint(1,num_samples+1,n)

# write x = a[i_j] + err_j for n/2 values then x = b[i_j] + err_j for n/2 valu
es
for j in range(0,int(n/2)):
    x[j] = a[i[j]-1] + err[j] # have to subtract 1 since python counts from 0
for j in range(int(n/2),n):
    x[j] = b[i[j]-1] + err[j]

# plot x as a scatter plot
axarr[1].scatter(x[:int(n/2),0],x[:int(n/2),1],marker='D',c='red')
axarr[1].scatter(x[int(n/2):n,0],x[int(n/2):n,1],marker='D',c='blue')
axarr[1].set_title('$x_j$, red datapoints classified as "0", blue datapoints c
lassified as "1"')

# save data for later tasks
task11a, task11b, task11x, task11y = a, b, x, y
```





Task 1.2

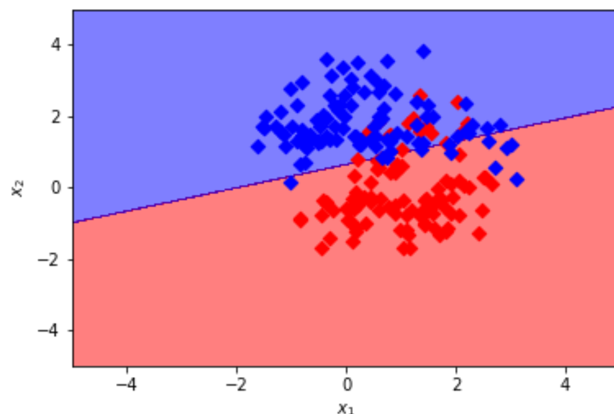
Implement a linear least squares algorithm using `numpy.linalg.solve`, then apply it to the data from Task 1.1. Specifically, we have to solve for α in the equation

$$X^T X\alpha = X^T y$$

where X is the matrix corresponding to the `x_j` from Task 1.1 and y corresponds to `y_j` from Task 1.1.

In [3]:
```
def lls(x,y):
    # pad x with ones on the left for the constant term in linear regression
    ones = np.ones(y.shape)
    x = np.concatenate((ones,x),axis = 1)
    # convert to x and y to matrices of floats
    x = np.mat(x).astype(float)
    y = np.mat(y).astype(float)
    return np.linalg.solve(x.T*x, x.T*y)

def f(x,alpha):
    x = np.array(x)
    # alpha should be an array so we can multiply row-wise (after a tranpose)
    alpha = np.array(alpha)
    a = alpha[1:].T*x
    return np.mat(alpha[0] + np.sum(a,axis=1)).T

def PlotContourLine(func, a, minx=-5, maxx=5, miny=-5, maxy=5, value=0):
    # This plots the contourline func(x) = value
    samplenum = 1000
    xrange = np.arange(minx, maxx, (maxx-minx)/samplenum)
    yrange = np.arange(miny, maxy, (maxy-miny)/samplenum)

    # This generates a two-dimensional mesh
    X, Y = np.meshgrid(xrange,yrange)

    argsForf = np.array([X.flatten(),Y.flatten()]).T
    Z = func(argsForf,a)
    Z = np.reshape(Z,X.shape)

    plt.xlim(minx, maxx)
    plt.ylim(miny, maxy)
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$x_2$')
    Z = np.where(Z > value, 1, -1)
    plt.contourf(X, Y, Z, alpha=0.5, colors=['r','b'])

alpha = lls(x,y)
PlotContourLine(f,alpha,value=1/2)
plt.scatter(x[:int(n/2),0],x[:int(n/2),1],marker='D',c='red')
plt.scatter(x[int(n/2):n,0],x[int(n/2):n,1],marker='D',c='blue')
```

Out[3]: <matplotlib.collections.PathCollection at 0x7f7b8c946390>



Task 1.3

Compute the confusion matrix C and compute its accuracy $\frac{\text{trace}(C)}{n}$, where $n = 200$ data points. We shift the indices of the confusion matrix back by one, so that for example C_{12} corresponds to the number of points classified as 0, where the real label is 1.

As for convention, we say that our function $f(x_1, x_2)$ classifies (x_1, x_2) to have label 0 when $f(x_1, x_2) \leq \frac{1}{2}$, and have label 1 when it is $> \frac{1}{2}$.

In [4]:
```
# function returns confusion matrix based on binary (0,1) classification data
def confusion_matrix(result, y): # result and y should be column vectors of eq
ual length
    # first column is test result, second is correct result
    data = pd.DataFrame(np.concatenate((result, y),1))
    C = np.empty([2,2])

    # data frame of test results = 0, and = 1
    result = data[data[0] <= 0.5], data[data[0] > 0.5]

    C[0,0] = result[0][result[0][1] == 0].shape[0]
    C[0,1] = result[0][result[0][1] == 1].shape[0]

    C[1,0] = result[1][result[1][1] == 0].shape[0]
    C[1,1] = result[1][result[1][1] == 1].shape[0]
    return C

result = f(x,alpha)
C = confusion_matrix(result, y)
accuracy = np.trace(C)/C.sum()
print('Confusion matrix = \n', C, '\nAccuracy = ', accuracy)
```

```
Confusion matrix =
 [[81. 11.]
 [19. 89.]]
Accuracy =  0.85
```

Task 1.4

We repeat task 1.3 but with 10,000 data points for each class (hence 20000 in total), using the same random samples a and b from task 1.1. In most cases we tend to observe that the accuracy here is slightly worse, though very close, compared to the accuracy achieved in task 1.3.

In [5]:

```
# number of datapoints
n = 20000

# initialise error, x, and y arrays
err = np.random.multivariate_normal(np.array([0,0]),cov/4,n)
x = np.zeros([n,2])
y = np.array(list(np.zeros([int(n/2),1])) + list(np.ones([int(n/2),1])))

# set up figures -- we have 2 figures stacked vertically
# fig, axarr = plt.subplots(2,1, figsize=(5,10))

# plot a and b on a scatter plot
# axarr[0].scatter(a[:,0],a[:,1],marker='D',c='red')
# axarr[0].scatter(b[:,0],b[:,1],marker='D',c='blue')
# axarr[0].set_title('$a$ and $b$')

# pick n equidistributed indices from 1,...,10
i = np.random.randint(1,num_samples+1,n)

# write x = a[i_j] + err_j for n/2 values then x = b[i_j] + err_j for n/2 valu
es
for j in range(0,int(n/2)):
    x[j] = a[i[j]-1] + err[j] # have to subtract 1 since python counts from 0
for j in range(int(n/2),n):
    x[j] = b[i[j]-1] + err[j]

# plot x as a scatter plot
# axarr[1].scatter(x[:int(n/2),0],x[:int(n/2),1],marker='D',c='red')
# axarr[1].scatter(x[int(n/2):n,0],x[int(n/2):n,1],marker='D',c='blue')
# axarr[1].set_title('$x_j$, red datapoints classified as "0", blue datapoints
classified as "1"')

alpha = lls(x,y)
# PlotContourLine(f,alpha,value=1/2)
# plt.scatter(x[:int(n/2),0],x[:int(n/2),1],marker='D',c='red')
# plt.scatter(x[int(n/2):n,0],x[int(n/2):n,1],marker='D',c='blue')

result = f(x,alpha)
C = confusion_matrix(result, y)

# accuracy
accuracy = np.trace(C)/C.sum()
print('Confusion matrix = \n', C, '\nAccuracy = ', accuracy)

# save data for later tasks
task14x, task14y = x, y
```

```
Confusion matrix =
 [[8391. 1231.]
 [1609. 8769.]]
Accuracy =  0.858
```

Task 1.5

With the aid of PANDAS, run the LLS algorithm on the Iris dataset.

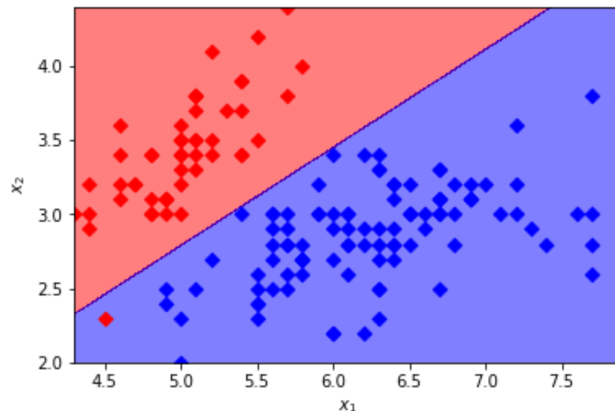
Observations

When we used only the first two features of the dataset as our training data for LLS learning, the accuracy of the separating hyperplane was highly dependent on what we were trying to classify. In the first instance, when we had *Iris-setosa* as one label and *Iris-versicolor* and *Iris-virginica* as the other label, we achieved ~99% accuracy. However, when we instead considered the task of classifying *Iris-versicolor* against *Iris-setosa* and *Iris-virginica*, the same algorithm gave us ~73% accuracy. The same is true when we used all features: ~100% accuracy when classifying *Iris-setosa* against the others, while ~73% accuracy when classifying *Iris-versicolor* against the others.

In [6]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.dat
a'
irisDataFrame = pd.read_csv(url, header=None)
```

In [7]:
```
# number of datapoints
n = irisDataFrame.shape[0]

# y_i = 0 for Iris-setosa and y_i = 1 for other two varieties; this info is st
ored in the column labeled '4'
irisDataFrame = irisDataFrame.sort_values(4) # make sure data is sorted by nam
e first (i.e. by label '4')
y = np.not_equal(irisDataFrame.as_matrix([4]),np.full([n,1],'Iris-setosa')).as
type(int) # convert bool to int

# run LLS algorithm on first two features only
x = irisDataFrame.as_matrix(range(0,2))
alpha0 = lls(x,y)

# for a nice graph, set maxx, maxy, etc based on data
maxx, minx, maxy, miny = np.max(x[:,0]), np.min(x[:,0]), np.max(x[:,1]), np.mi
n(x[:,1])

n_sentosa = np.sum(1-y) # the number of values labeled '0', i.e. number of sen
tosa (which are all listed first)
PlotContourLine(f,alpha0, maxx=maxx, minx=minx, maxy=maxy, miny=miny, value=1/
2)
plt.scatter(x[:n_sentosa,0],x[:n_sentosa,1],marker='D',c='red')
plt.scatter(x[n_sentosa:n,0],x[n_sentosa:n,1],marker='D',c='blue')

result = f(x,alpha0)
C = confusion_matrix(result, y)
accuracy = np.trace(C)/C.sum()
print('LLS for first 2 features','\nConfusion matrix = ',C,'\nAccuracy =',accu
racy)
```

```
LLS for first 2 features
Confusion matrix =  [[ 49.   0.]
 [  1. 100.]]
Accuracy = 0.9933333333333333
```

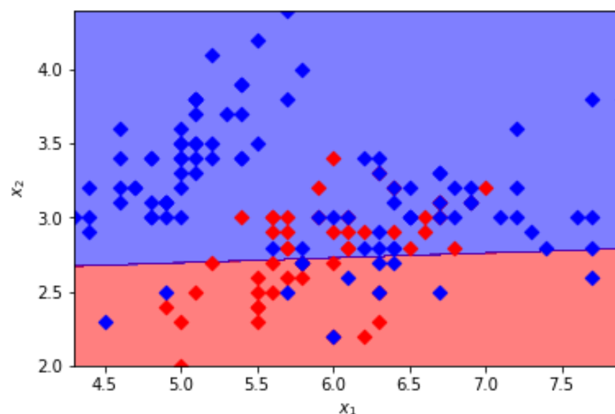


In [8]:
```
# run LLS algorithm on all features
x = irisDataFrame.as_matrix(range(0,4))
alpha1 = lls(x,y)
result = f(x,alpha1)
C = confusion_matrix(result, y)
accuracy = np.trace(C)/C.sum()
print('LLS for all features', '\nConfusion matrix = ',C,'\nAccuracy =',accurac
y)
```

```
LLS for all features
Confusion matrix =  [[ 50.   0.]
 [  0. 100.]]
Accuracy = 1.0
```

In [9]:
```
# to ignore the chain assignment warning a few lines below
pd.options.mode.chained_assignment = None  # default='warn'

# first process irisDataFrame so that 'Iris-versicolor' is labeled 0, the othe
r two types are labeled 1
df = irisDataFrame.copy()
labels = df.loc[:,4]
labels[df.loc[:,4] == 'Iris-versicolor'] = 0
labels[df.loc[:,4].isin(['Iris-setosa','Iris-virginica'])] = 1
df = df.sort_values(4) # now versicolor is listed first
y = df.as_matrix([4])

# run LLS algorithm on first two features only
x = df.as_matrix(range(0,2))
alpha0 = lls(x,y)

# for a nice graph, set maxx, maxy, etc based on data
maxx, minx, maxy, miny = np.max(x[:,0]), np.min(x[:,0]), np.max(x[:,1]), np.mi
n(x[:,1])

n_versicolor = np.sum(1-y) # the number of values labeled '0', i.e. number of
versicolor (which are all listed first)
PlotContourLine(f,alpha0, maxx=maxx, minx=minx, maxy=maxy, miny=miny, value=1/
2)
plt.scatter(x[:n_versicolor,0],x[:n_versicolor,1],marker='D',c='red')
plt.scatter(x[n_versicolor:n,0],x[n_versicolor:n,1],marker='D',c='blue')

result = f(x,alpha0)
C = confusion_matrix(result, y)
accuracy = np.trace(C)/C.sum()
print('LLS for first 2 features','\nConfusion matrix = ',C,'\nAccuracy =',accu
racy)
```

```
LLS for first 2 features
Confusion matrix =  [[21. 12.]
 [29. 88.]]
Accuracy = 0.7266666666666667
```



In [10]:
```
# run LLS algorithm on all features
x = df.as_matrix(range(0,4))
alpha1 = lls(x,y)
result = f(x,alpha1)
print(result.shape, type(result))
C = confusion_matrix(result,y)
accuracy = np.trace(C)/C.sum()
print('LLS for all features', '\nConfusion matrix = ',C,'\nAccuracy =',accurac
y)
```

```
(150, 1) <class 'numpy.matrixlib.defmatrix.matrix'>
LLS for all features
Confusion matrix =  [[24. 14.]
 [26. 86.]]
Accuracy = 0.7333333333333333
```

Task 1.6

Implement gradient descent and run an LLS algorithm on the data from a.1 of task 1.5. We have that

$$J(\boldsymbol{\alpha}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x_i}\boldsymbol{\alpha} - y_i)^2 = \frac{1}{n}(\mathbf{X}\boldsymbol{\alpha} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\alpha} - \mathbf{y})$$

$$\frac{\partial J}{\partial \alpha_j} = \frac{2}{n}\sum_{i=1}^{n}(\mathbf{x_i}\boldsymbol{\alpha} - y_i)\,x_{ij}$$

$$\nabla J(\boldsymbol{\alpha}) = \left[\frac{\partial J}{\partial \alpha_1} \cdots \frac{\partial J}{\partial \alpha_m}\right] = \frac{2}{n}(\mathbf{X}\boldsymbol{\alpha} - \mathbf{y})^T\mathbf{X}$$

where x_{ij} denotes the jth component of the (row) vector $\mathbf{x_i}$.

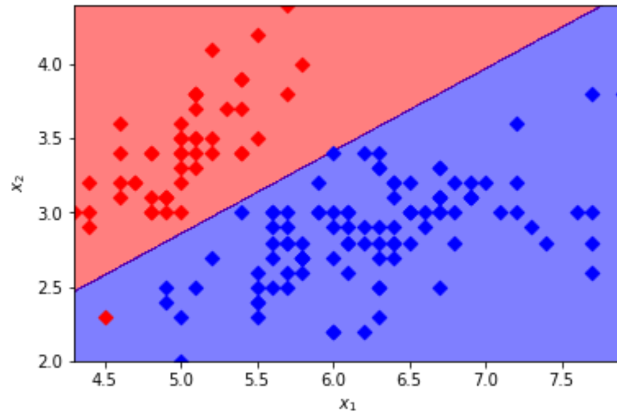
Observations

Using a stepwidth of 0.01, we achieved convergence. In this case, the cost of J is drastically reduced in the first ~200 steps, and then proceeds to asymptotically decrease.

In [11]:

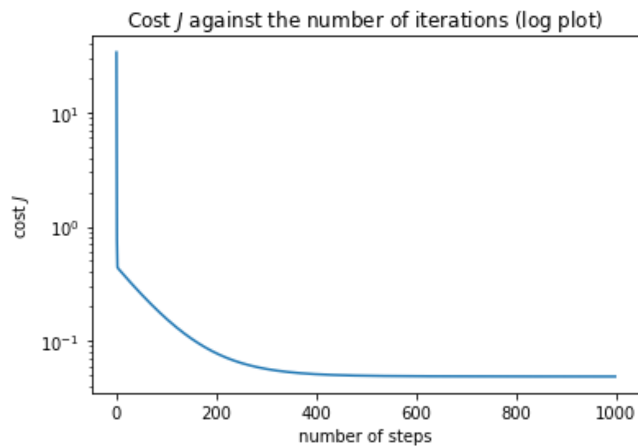
```
# load data from task 1.5, part a.1
y = np.not_equal(irisDataFrame.as_matrix([4]),np.full([n,1],'Iris-setosa')).as
type(int) # convert bool to int
x = irisDataFrame.as_matrix(range(0,2))

def lls_gd(x,y, stepwidth=0.01, maxSteps=1000, accuracyThreshold=0.0001):
    # pad x with ones on the left for the constant term in linear regression
    ones = np.ones(y.shape)
    x = np.concatenate((ones,x),axis = 1)
    # convert x and y to matrices of floats
    x = np.mat(x).astype(float)
    y = np.mat(y).astype(float)
    m = x.shape[1] # number of features including the 1's

    #initialise alpha (here known as 'a') randomly, our (iterated) solution
    a = np.asmatrix(np.random.random([m,1]))
    step = 0

    # initialise data array for J vs steps
    iterData = np.empty([maxSteps, 2])

    # returns value of cost function J
    def J(x,y,a):
        c = (x*a - y).T*(x*a-y)
        return c/n

    # returns hard-coded gradient
    def gradient(x,y,a):
        g = (x*a - y).T*x
        return 2*g.T/n # while the gradient is a row vector, in our applicatio
n we expect a column vector

    while (np.sum(gradient(x,y,a)) > accuracyThreshold) and (step < maxSteps):
        iterData[step, 0], iterData[step, 1] = step, J(x,y,a)
        a = a - stepwidth * gradient(x,y,a)
        step += 1
    return a, iterData[0:step, :]

alpha, iterData = lls_gd(x,y)

# for a nice graph, set maxx, maxy, etc based on data
maxx, minx, maxy, miny = np.max(x[:,0]), np.min(x[:,0]), np.max(x[:,1]), np.mi
n(x[:,1])

n_versicolor = np.sum(1-y) # the number of values labeled '0', i.e. number of
versicolor (which are all listed first)
PlotContourLine(f,alpha, maxx=maxx, minx=minx, maxy=maxy, miny=miny, value=1/2
)
plt.scatter(x[:n_versicolor,0],x[:n_versicolor,1],marker='D',c='red')
plt.scatter(x[n_versicolor:n,0],x[n_versicolor:n,1],marker='D',c='blue')

result = f(x,alpha)

C = confusion_matrix(result, y)
accuracy = np.trace(C)/C.sum()
print('LLS for all features', '\nConfusion matrix = ',C,'\nAccuracy =',accurac
y)
```

```
LLS for all features
Confusion matrix =  [[ 49.   0.]
 [  1. 100.]]
Accuracy = 0.9933333333333333
```



In [12]:

```
# Plot cost function J against number of steps
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('Cost $J$ against the number of iterations (log plot)')
ax.set_xlabel('number of steps')
ax.set_ylabel('cost $J$')
ax.set_yscale('log')
ax.plot(iterData[:,0], iterData[:,1])

# save iterData for the next graph for comparison
iterData_unnormalised = iterData
```



Task 1.7

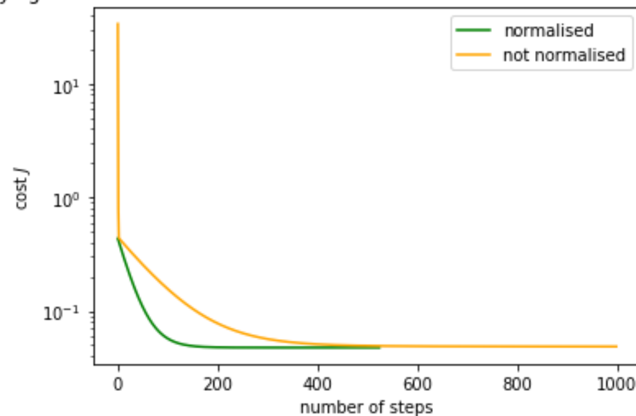
Normalise the data from task 1.5 (a.1), and then apply the same gradient descent-least squares algorithm.

Observation

As compared to the case previously with un-normalised data, we see that the algorithm begins with a lower cost J, and approaches convergence much quicker, with the initial radical decrease in cost occuring from step 1 to ~100.

In [13]:
```
irisDataFrameNormalised = irisDataFrame.copy()
# mu[i] is the mean of the ith feature
mu = irisDataFrame.mean(0).as_matrix()
# sd[i] is the std deviation of the ith feature
sd = irisDataFrame.std(0).as_matrix()
# irisDataFrameNormalised is irisDataFrame with normalised values

for feature in range(0,4):
    irisDataFrameNormalised[feature] = (irisDataFrameNormalised[feature] - mu[
feature])/sd[feature]

# run lls_gd on normalised data
y = np.not_equal(irisDataFrameNormalised.as_matrix([4]),np.full([n,1],'Iris-se
tosa')).astype(int) # convert bool to int
x = irisDataFrameNormalised.as_matrix(range(0,2))

alpha, iterData = lls_gd(x,y)

# Plot cost function J against number of steps
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('Cost $J$ against the number of iterations with both raw and norm
alised data (log plot)')
ax.set_xlabel('number of steps')
ax.set_ylabel('cost $J$')
ax.set_yscale('log')
normalised_line = ax.plot(iterData[:,0], iterData[:,1],color='green')
unnormalised_line = ax.plot(iterData_unnormalised[:,0], iterData_unnormalised[
:,1],color='orange')
ax.legend((normalised_line[0], unnormalised_line[0]), ('normalised', 'not norm
alised'))
```

Out[13]: <matplotlib.legend.Legend at 0x7f7b8c77a048>



Task 1.8

Implement the k-nearest neighbour algorithm. **By convention**, $\frac{1}{2}$ is classified to be 0, which is the behaviour of `np.round`.

We apply the algorithm to the data of task 1.1, which we will use as both training and test data.

A speed-up

If we assume that we are computing the nearest neighbour for multiple values for k, where $k = 1, 2, 3, m$, the following is a possible speedup (since the code can be made more "specific", really):

Observe that

$$\text{nearestNeighbour}_{k+1}(x) = \frac{k}{k+1}\text{nearestNeighbour}_k(x) + \frac{1}{k+1}y_{k+1}$$

where y_{k+1} is the training result for the training data point that is the closest to x out of the set $N_{k+1}(x)$, or in other words the closest point from $N_m(x) \setminus N_k(x)$. It may be that this is faster since we are computing the averages and sorting while assigning.

Unknown bug

While `nearNeighRoutineConsecutive` runs a lot faster than `nearNeighRoutine`, it appears that the two give slightly different results. Sometimes the former is slightly more accuracy, sometimes the latter is slightly more accurate (empirically, within 1% of each other for the training data from 1.1). This may be due to sensitivity of the computations leading to some rounding error.

In [14]:

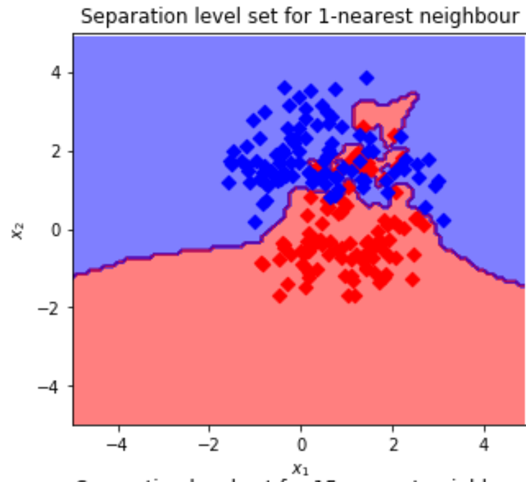
```
import time
from IPython.display import clear_output

training_features, training_results = task11x, task11y

# number of test data points and features
n = x.shape[0]
num_features = x.shape[1]

# # computes the nearest neighbour of a single point x from a collection of tr
aining data 'training'
# def nearNeigh(k, x, training_features, training_results): # x ought to be a
row vector, training is a matrix
#     # if k > the number of data points in y, set k to be the length of y
#     if k > training_features.shape[0]:
#         k = training_features.shape[0]

#     distances = sp.distance.cdist(x, training_features).T
#     data = pd.DataFrame(np.concatenate((distances,training_results),1))

#     # sorting based on the distance (leftmost) column, ascending order by ro
w; second column is the training results
#     y = data.sort_values(0)[1].as_matrix()
# #     print(y[0:k])
#     return np.sum(y[0:k])/k

# def nearNeighRoutine(k, x, training_features, training_results):
#     x = np.mat(x) # in case x is a single datapoint
#     m = x.shape[0] # number of test data points
#     results = np.empty([m,1])
#     for i in range(0,m):
# #         print(nearNeigh(k,x[i],training_features,training_results))
#         results[i] = np.round(nearNeigh(k,x[i],training_features,training_re
sults))
#     return results

def nearNeighRoutineConsecutive(k_end, x, training_features, training_results,
timer=0):
    # we assume k begins at 1. k_end must be a positive integer
    if k_end < 1:
        raise Error('The final value for k must be a positive integer')

    x = np.mat(x) # in case x is a single datapoint
    m = x.shape[0] # number of test data points

    # initialise results array: results[i][j] will be the nearest neighbour su
m of x[i] when k = j
    results = np.empty((m, k_end))

    # since we are running in consecutive mode, we start with i = 1, k = 1
    for i in range(m):
        start = time.monotonic()
        distances = sp.distance.cdist(x[i], training_features).T
        very_big_value = np.max(distances) + 1
        nearest = distances.argmin()
        results[i][0] = training_results[nearest] # when k = 1, the nearest ne
ighbour weighted sum is just the nearest point
        distances[nearest] = very_big_value # will no longer be returned as th
e minimum in future argmin() steps
        ### begin timer code ###
        if timer == 1:
            end = time.monotonic()
            print('x[{0}] was computed in {1} seconds'.format(i, end-start))
```

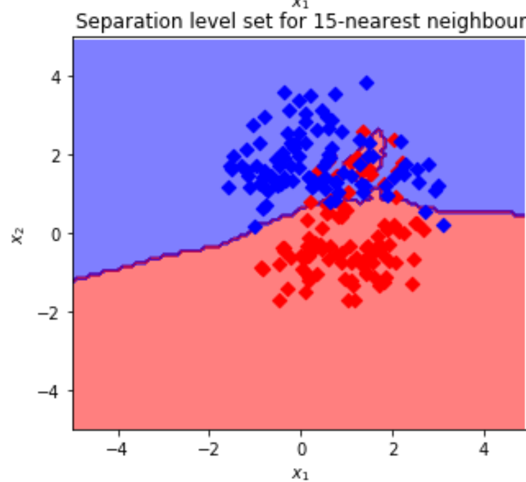
```
1-nearest for task 1.1
Confusion matrix = [[100.   0.]
 [  0. 100.]]
Accuracy = 1.0
15-nearest for task 1.1
Confusion matrix = [[83. 10.]
 [17. 90.]]
Accuracy = 0.865
30-nearest for task 1.1
Confusion matrix = [[81.  6.]
 [19. 94.]]
Accuracy = 0.875
```

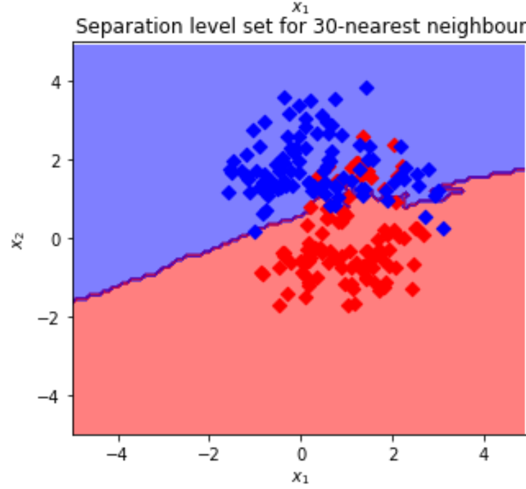
In [15]:

```
# Scatter plots of k-nearest neighbour for k = 1, 15, 30

# set up figures -- we have 3 figures stacked vertically
fig, axarr = plt.subplots(3,1, figsize=(5,15))

def PlotContourLineNearNeigh(axis_arr, func, kays, training_features, training
_results, minx=-5, maxx=5, miny=-5, maxy=5, value=0):
    if axis_arr.shape[0] != len(kays):
        raise Error('the number of given matplot axes does not match the numbe
r of k\'s to be used with NearNeigh')

    # This plots the contourline func(x) = value
    samplenum = 100
    xrange = np.arange(minx, maxx, (maxx-minx)/samplenum)
    yrange = np.arange(miny, maxy, (maxy-miny)/samplenum)

    kays.sort()
    num_kays, k = len(kays), kays[-1]

    # This generates a two-dimensional mesh
    X, Y = np.meshgrid(xrange,yrange)

    argsForf = np.array([X.flatten(),Y.flatten()]).T
    results = func(kays, argsForf, training_features, training_results,timer=2
)

    for i in range(num_kays):
        Z = np.reshape(results[:,i],X.shape)
        axis_arr[i].set_xlim(minx, maxx)
        axis_arr[i].set_ylim(miny, maxy)
        axis_arr[i].set_xlabel(r'$x_1$')
        axis_arr[i].set_ylabel(r'$x_2$')
        Z = np.where(Z > value, 1, -1)
        axis_arr[i].contourf(X, Y, Z, alpha=0.5, colors=['r','b'])
        axis_arr[i].scatter(training_features[0:100,0],training_features[0:100
,1],marker='D',c='red')
        axis_arr[i].scatter(training_features[100:200,0],training_features[100
:200,1],marker='D',c='blue')
        axis_arr[i].set_title('Separation level set for ' + str(kays[i]) + '-n
earest neighbour')

# in training_features, first 100 are labeled 0 (red) and second 100 are 1 (bl
ue)
kays = [1, 15, 30] # values for k
PlotContourLineNearNeigh(axarr, nearNeighRoutine, kays, training_features, tra
ining_results)
```

```
100.0% completed
```







Task 1.9

Run the k-nearest neighbour algorithm for all possible k on the data from 1.1 and then from 1.4.

Observations

Excluding the case $k = 1$ when we used task 1.1 data (which gives 100% accuracy, since the test and training data are the same), we see that the accuracy against k is very similar between the use of task 1.1 and task 1.4 data for k-nearest neighbour. A pattern between accuracy and k is clearer in the case of using task 1.4 data: there is an overall decrease in accuracy as k increases, though not by much (excluding $k = 200$). We cannot seem to explain the sudden drop to 50% accuracy for $k = 200$ when using task 1.4 data, but it appears that 50 accuracy persists for $k \geq 200$ in that case.

In [16]:
```
# load task 1.1 data as test data
test_features, test_results = task11x, task11y

# save the accuracy data for 1.1
accuracies11 = np.zeros([200,1])

# kays = list(range(1,200+1))
results11 = nearNeighRoutineConsecutive(200,test_features,training_features,tr
aining_results,timer=2)
```

```
100.0% completed
```

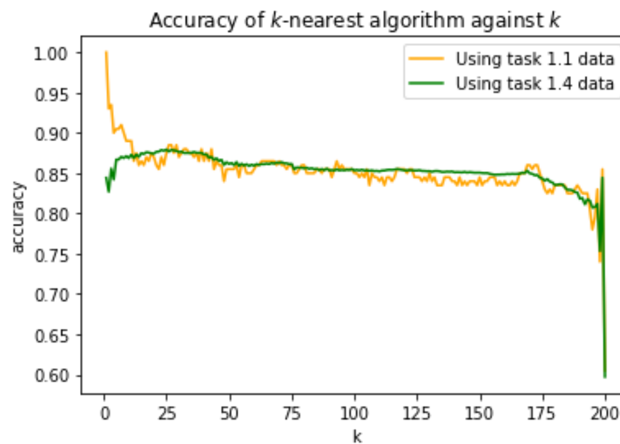
In [17]:
```
# load task 1.4 data as test data
test_features, test_results = task14x, task14y

# save the accuracy data for 1.4
accuracies14 = np.zeros([200,1])

# kays = list(range(1,200+1))
results = nearNeighRoutineConsecutive(200,test_features,training_features,trai
ning_results,timer=2)
```

```
100.0% completed
```

In [18]:
```
# compute confusion matrices
for i in range(200):
    C = confusion_matrix(results11[:,i:i+1], task11y) # task 1.1 data
    accuracies11[i] = np.trace(C)/C.sum()
for i in range(200):
    C = confusion_matrix(results[:,i:i+1], task14y) # task 1.4 data
    accuracies14[i] = np.trace(C)/C.sum()
```

In [19]:
```
# plot accuracy data against k value

# set up figures -- we have 2 figures stacked vertically
fig, ax = plt.subplots()

ax.set_title('Accuracy of $k$-nearest algorithm against $k$')
ax.set_xlabel('k')
ax.set_ylabel('accuracy')
task11line = ax.plot(np.arange(1,200+1), accuracies11[:], color='orange')
task14line = ax.plot(np.arange(1,200+1), accuracies14[:], color='green')
ax.legend((task11line[0], task14line[0]), ('Using task 1.1 data', 'Using task 1.4 data'))
```

Out[19]: <matplotlib.legend.Legend at 0x7f7b8c880198>



In [20]:
```
accuracies14.argmax()
```

Out[20]: 24