Classification Using LDA, QDA, and Linear and Nonlinear Logistic Regression

Swetha Varadarajan

November 14, 2014

Contents

Abstract

Classification of data becomes an essential tool in analysing various parameters and finds a critical role in every application. In this report, we study classification of data using four different methods.These methods differ in the decision boundary and shape taken for classification. This is demonstrated on a simple one-dimensional as well as the real data sample taken from [2]. The results are observed and discussed.

1 Introduction

Classification when viewed as a problem in machine learning, involves the data set to be divided into K discrete classes Ck where k=1,2,k. These classes are also called as decision regions whose boundaries are called as decision boundaries or decision surfaces[1]. This report involves four methods namely Linear Discriminant Analysis(LDA),Quadratic Discriminant Analysis(QDA), Linear Logistic regression(LLR) and Nonlinear Logistic Regression (NLR). LDA and QDA comes under the category of Discriminant Analysis wherein a discriminant function is used to classify the data set into dis-joint classes.

1. **LDA:** Decision boundary is linear. The discriminant function for LDA is,

$$\delta_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + \log P(C = k) \tag{1}$$

2. **QDA:** Decision boundary is quadratic. The Discriminant function is for the Quadratic Discriminant Analysis is,
$$\delta_k(x) = -\frac{1}{2}\ln|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T \tag{2}$$

3. **LLR:** LDA introduces masking problem. In order to avoid this, a likelihood based decision boundary is created and solved using scaled conjugate gradients. The log likelihood is given by
$$LL(\beta) = logL(\beta) = \sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k} * logp(C=k|x_n) \tag{3}$$

4. **NLR:** Neural network based classification. Useful for non-linear data samples.

The remaining report is organized as follows. Section 2 describes the experiments on one-dimensional data. Section 3 talks about real data sample selected and the classification performed. Section 4 is for additional question followed by conclusion and references.

2 One-dimensional data

This section describes the python code and the experimental results obtained on one-dimensional data for the 4 classifying methods selected.Before going into the codes, the reading or preparation of training data is common for all the 4 methods which is shown in the in the following listing.

```
D=1 #dimension
N=10 #number of samples in each class
X1=np.random.normal(1.0,0.1,(N,D))
T1=np.array([1]*N).reshape((N,1))
X2=np.random.normal(2.0,0.1,(N,D))
T2=np.array([2]*N).reshape((N,1))
X3=np.random.normal(3.0,0.1,(N,D))
T3=np.array([3]*N).reshape((N,1))
data=np.hstack((np.vstack((X1,X2,X3)),np.vstack((T1,T2,T3))))
X=data[:,0:D]
T=data[:,-1]
standardize,_=makeStandardizeF(X)
Xs=standardize(X)
```

In the above code three classes of data with 10 samples each has been created. Each class has a standard deviation of 0.1 and mean of 1,2,3 respectively. All the classes of data are combined to obtain the train data. Finally the data is divided into input (X) and target(T). The input data(X) is standardized using the makeStandardizeF() function. The code for makeStandardizeF function is shown below:

```
def makeStandardizeF(X):
        means = X.mean(axis=0)
        stds = X.std(axis=0)
        def standardize(origX):
            return (origX - means) / stds
        def unStandardize(stdX):
            return stds * stdX + means
        return (standardize, unStandardize)
```

2.1 QDA method

The python code is shown in the following listing.

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

```

```python
#standardization function
def makeStandardize(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)

#QDA function
def discQDA(X, standardize, mu, Sigma, prior):
    Xc = standardize(X) - mu
    if Sigma.size == 1:
        Sigma = np.asarray(Sigma).reshape((1,1))
    det = np.linalg.det(Sigma)
    if det == 0:
        raise np.linalg.LinAlgError('discQDA(): Singular covariance matrix')
    SigmaInv = np.linalg.inv(Sigma)     # pinv in case Sigma is singular
    return -0.5 * np.log(det) - 0.5 * np.sum(np.dot(Xc,SigmaInv)
            * Xc, axis=1)   + np.log(prior)

# Normald function
def normald(X, mu=None, sigma=None):
    d = X.shape[1]
    if np.any(mu == None):
        mu = np.zeros((d,1))
    if np.any(sigma == None):
        sigma = np.eye(d)
    detSigma = sigma if d == 1 else np.linalg.det(sigma)
    if detSigma == 0:
        raise np.linalg.LinAlgError('normald(): Singular covariance matrix')
    sigmaI = 1.0/sigma if d == 1 else np.linalg.inv(sigma)
    normConstant = 1.0 / np.sqrt((2*np.pi)**d * detSigma)
    diffv = X - mu.T # change column vector mu to be row vector
    return normConstant * np.exp(-0.5 * np.sum(np.dot(diffv, sigmaI)
     * diffv, axis=1))[:,np.newaxis]

#Training data generation
D=1 #dimension = 1
N=10 #number of samples in each class
X1=np.random.normal(1.0,0.1,(N,D))
T1=np.array([1]*N).reshape((N,1))
X2=np.random.normal(2.0,0.1,(N,D))
T2=np.array([2]*N).reshape((N,1))
X3=np.random.normal(3.0,0.1,(N,D))
T3=np.array([3]*N).reshape((N,1))
data=np.hstack((np.vstack((X1,X2,X3)),np.vstack((T1,T2,T3))))
X=data[:,0:D]
T=data[:,-1]
standardize,_=makeStandardize(X)
Xs=standardize(X)

#Parameter calculation
class1rows=T==1
class2rows=T==2
class3rows=T==3

mu1=np.mean(Xs[class1rows,:],axis=0)
```

```
mu2=np.mean(Xs[class2rows,:], axis=0)
mu3=np.mean(Xs[class3rows,:], axis=0)

Sigma1=np.cov(Xs[class1rows,:].T)
Sigma2=np.cov(Xs[class2rows,:].T)
Sigma3=np.cov(Xs[class3rows,:].T)

N1=np.sum(class1rows)
N2=np.sum(class2rows)
N3=np.sum(class3rows)

N=len(T)
prior1=N1/float(N)
prior2=N2/float(N)
prior3=N3/float(N)

#Testing data creation
nNew = 100
newData = np.linspace(0,4,nNew).repeat(D).reshape((nNew,D))

#Model building
d1=discQDA(newData,standardize,mu1,Sigma1,prior1)
d2=discQDA(newData,standardize,mu2,Sigma2,prior2)
d3=discQDA(newData,standardize,mu3,Sigma3,prior3)

#Plots
plt.figure(figsize=(10,10))

#code for plotting between classes and data
plt.subplot(6,1,1)
plt.plot(X[class1rows],T[class1rows],"ro")
plt.plot(X[class2rows],T[class2rows],"go")
plt.plot(X[class3rows],T[class3rows],"bo")
plt.ylabel("class values")

#code for plotting probabilities
plt.subplot(6,1,2)
newDataS = standardize(newData)
probs = np.hstack((normald(newDataS,mu1,Sigma1),
normald(newDataS,mu2,Sigma2),normald(newDataS,mu3,Sigma3)))
plt.plot(newData[:,0],probs)
plt.ylabel("QDA P(x|Class=k)\n from disc funcs", multialignment="center")

#code for plotting the curve p(x) for the test data
plt.subplot(6,1,3)
p1= normald(newDataS,mu1,Sigma1)
p2= normald(newDataS,mu2,Sigma2)
p3= normald(newDataS,mu3,Sigma3)
px=p1*prior1+p2*prior2+p3*prior3
plt.plot(newData,px)
plt.ylabel("p(x)");

#code for plotting the curve p(c=k|x)
pofc1=p1*prior1/px
pofc2=p2*prior2/px
pofc3=p3*prior3/px
plt.subplot(6,1,4)
plt.plot(newData,np.hstack((pofc1,pofc2,pofc3)))
plt.ylabel("p(c=k|x)")
```

```
122
123 #code for plotting the discriminants
124 plt.subplot(6,1,5)
125 plt.plot(newDataS,d1,"b")
126 plt.plot(newDataS,d2,"g")
127 plt.plot(newDataS,d3,"y")
128 plt.ylabel("QDA discriminants");
129
130 #code for plotting  the class predicted by the classifier for the test data.
131 plt.subplot(6,1,6)
132 preTest = np.argmax(np.vstack((d1,d2,d3)),axis=0)
133 plt.plot(newData,preTest,'o')
134 plt.ylabel("Predicted class by QDA")
135
136 #plot saving
137 plt.subplots_adjust(hspace=0.5,wspace = .5)
138 plt.savefig('qdatoy.png')
```

There are 10 parts in the code.

1. Lines[1-2]: Normal initialization of packages
2. Lines[4-12]: Standardization function definition
3. Lines[14-24]: QDA function. Returns equation 2 taking the model parameters, standardize function and the input data as input parameters.
4. Lines[26-40]: Normald function used to calculate the normal distribution given the mean and variance.
5. Lines[42-55]: Training data generation as discussed earlier.
6. Lines[57-77]: Model parameter calculation. Calculates sigma, mean and prior class probabilities for all the 3 classes.
7. Lines[79-81]: 100 samples of test data creation.
8. Lines[83-86]: Calling the QDA function with the parameter set defined above.
9. Lines[88-134]: Plotting section for the six plots. Plot-1 is nothing but training data versus their respective classes. Plot-5 is nothing but the plot of the returned discriminant values. Plot-6 is test data versus predicted classes which is determined by taking argument max of the discriminant function obtained for each sample with respect to the 3 classes.The probabilities p(x—C=k), p(x),p(C=k—x) are calculated and plotted in plots-2,3,4. p(x—C=k) is calculated using the normald function. p(x) is calculated by using the formula p(x—C=k)*p(Class=k). p(C=k—x) is calculated using the formula p(x)*p(x—C=k).
10. Lines[136-138]: Saving the plot with spacing adjustments between the sub-plots.

The plots are shown in figure-1. The training data has 3 different classes each of which is shown in different color in the first sub-plot. The second plot is nothing but the 3 normal distributions. these don't overlap because we have taken the means as 1,2 and 3.The 3rd plot shows the height of p(x) to be around 1. This is because, from the probability formula, we divide each P(x—k) by its prior probability which is 1/3 for all the three classes. Hence we can observe the difference in the heights of graph 2 and 3. The graph 4 tells us that the probability of the sum of the classes should always be 1. For instance at point 1.6 in x-axis, we can see that red line is almost zero and there is a intersection of blue and green line at the center(0.5 with respect to y-axis). Hence, this sum up to 1. The fifth graph is the discriminant. It can be seen that everything is a parabola hence proving it to be quadratic in nature. It can be seen that yellow and blue pass over the green parabola proving that the modelling has been done quite successfully and the classification takes place according to the argmax of the three values. From the final graph one can observe how the testing data is divided into three classes.

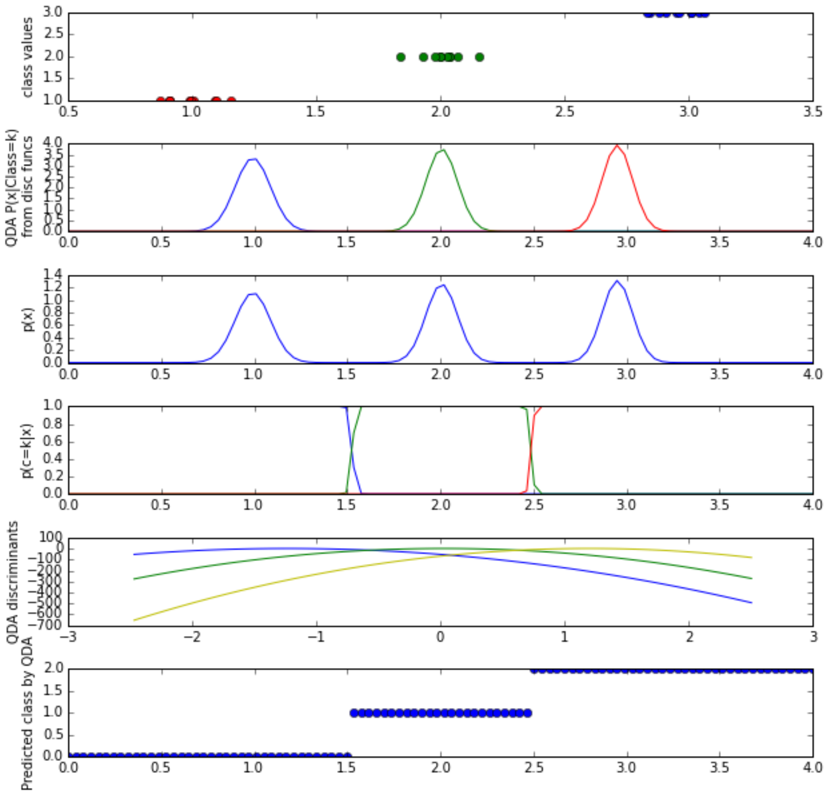

Figure 1: QDA-1D data

2.2 LDA method

The python code is shown in the listing below

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

#standardization function
def makeStandardize(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0)
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)

#LDA function
def discLDA(X, standardize ,mu, Sigma, prior):
    Xc=standardize(X)-mu
    if Sigma.size==1:
        Sigma=np.asarray(Sigma).reshape((1,1))
    det=np.linalg.det(Sigma)
    if det==0:
        raise np.linalg.LinAlgError('discLDA(): Sigular covarience matrix')
    SigmaInv=np.linalg.inv(Sigma)
    return np.dot(X,np.dot(SigmaInv,mu))-0.5*np.dot(np.dot(Xc,
                        SigmaInv),mu)+np.log(prior)

#Normald function
def normald(X, mu=None, sigma=None):
    d = X.shape[1]
    if np.any(mu == None):
        mu = np.zeros((d,1))
    if np.any(sigma == None):
        sigma = np.eye(d)
    detSigma = sigma if d == 1 else np.linalg.det(sigma)
    if detSigma == 0:
        raise np.linalg.LinAlgError('normald(): Singular covariance matrix')
    sigmaI = 1.0/sigma if d == 1 else np.linalg.inv(sigma)
    normConstant = 1.0 / np.sqrt((2*np.pi)**d * detSigma)
    diffv = X - mu.T # change column vector mu to be row vector
    return normConstant * np.exp(-0.5 * np.sum(np.dot(diffv, sigmaI)
     * diffv, axis=1))[:,np.newaxis]

#Training data generation
D=1 #dimension = 1
N=10 #number of samples in each class
X1=np.random.normal(1.0,0.1,(N,D))
T1=np.array([1]*N).reshape((N,1))
X2=np.random.normal(2.0,0.1,(N,D))
T2=np.array([2]*N).reshape((N,1))
X3=np.random.normal(3.0,0.1,(N,D))
T3=np.array([3]*N).reshape((N,1))
data=np.hstack((np.vstack((X1,X2,X3)),np.vstack((T1,T2,T3))))
X=data[:,0:D]
T=data[:,-1]
standardize,_=makeStandardize(X)
Xs=standardize(X)

```

```
#Parameter calculation
class1rows=T==1
class2rows=T==2
class3rows=T==3

mu1=np.mean(Xs[class1rows,:], axis=0)
mu2=np.mean(Xs[class2rows,:], axis=0)
mu3=np.mean(Xs[class3rows,:], axis=0)

#calculating linear sigma for LDA
#Note the difference from QDA equivalent
Sigma=prior1 * Sigma1 + prior2 * Sigma2 + prior3 * Sigma3

N1=np.sum(class1rows)
N2=np.sum(class2rows)
N3=np.sum(class3rows)

N=len(T)
prior1=N1/float(N)
prior2=N2/float(N)
prior3=N3/float(N)

#Testing data creation
nNew = 100
newData = np.linspace(0,4,nNew).repeat(D).reshape((nNew,D))

#building model
dl1=discLDA(newData,standardize,mu1,Sigma,prior1)
dl2=discLDA(newData,standardize,mu2,Sigma,prior2)
dl3=discLDA(newData,standardize,mu3,Sigma,prior3)

#plot codes
plt.figure(figsize=(10,10))

#code for plotting between classes and data
plt.subplot(6,1,1)
plt.plot(X[class1rows],T[class1rows],"ro")
plt.plot(X[class2rows],T[class2rows],"go")
plt.plot(X[class3rows],T[class3rows],"bo")
plt.ylabel("class values")

#code for plotting probabilities
plt.subplot(6,1,2)
newDataS = standardize(newData)
probs = np.hstack((normald(newDataS,mu1,Sigma),normald(newDataS,mu2,Sigma)
,normald(newDataS,mu3,Sigma)))
plt.plot(newData[:,0],probs)
plt.ylabel("LDA P(x|Class=k)\n from disc funcs", multialignment="center")

#code for plotting the curve p(x) for the test data
plt.subplot(6,1,3)
p1= normald(newDataS,mu1,Sigma)
p2= normald(newDataS,mu2,Sigma)
p3= normald(newDataS,mu3,Sigma)
px=p1*prior1+p2*prior2+p3*prior3
plt.plot(newData,px)
plt.ylabel("p(x)");

#code for plotting the curve p(c=k|x)
```

```
116 pofc1=p1*prior1/px
117 pofc2=p2*prior2/px
118 pofc3=p3*prior3/px
119 plt.subplot(6,1,4)
120 plt.plot(newData,np.hstack((pofc1,pofc2,pofc3)))
121 plt.ylabel("p(c=k|x)")
122
123 #code for plotting the discriminants
124 plt.subplot(6,1,5)
125 plt.plot(newDataS,dl1,"b")
126 plt.plot(newDataS,dl2,"g")
127 plt.plot(newDataS,dl3,"y")
128 plt.ylabel("LDA discriminants");
129
130 #code for plotting  the class predicted by the classifier for the test data.
131 plt.subplot(6,1,6)
132 preTest = np.argmax(np.vstack((dl1,dl2,dl3)),axis=0)
133 plt.plot(newData,preTest,'o')
134 plt.ylabel("Predicted class by LDA")
135
136 #code for saving the plots
137 plt.subplots_adjust(hspace=0.5,wspace = .5)
138 plt.savefig('ldatoy.png')
```

There are 10 parts in the code.

1. Lines[1-2]: Normal initialization of packages
2. Lines[4-12]: Standardization function definition
3. Lines[14-24]: LDA function. Returns equation 1 taking the model parameters, standardize function and the input data as input parameters.
4. Lines[26-40]: Normald function used to calculate the normal distribution given the mean and variance.
5. Lines[42-55]: Training data generation as discussed earlier.
6. Lines[57-77]: Model parameter calculation. Calculates sigma, mean and prior class probabilities for all the 3 classes. Difference from QDA's sigma calculation.
7. Lines[79-81]: 100 samples of test data creation.
8. Lines[83-86]: Calling the LDA function with the parameter set defined above.
9. Lines[88-134]: Plotting section for the six plots. Plot-1 is nothing but training data versus their respective classes. Plot-5 is nothing but the plot of the returned discriminant values. Plot-6 is test data versus predicted classes which is determined by taking argument max of the discriminant function obtained for each sample with respect to the 3 classes.The probabilities p(x—C=k), p(x),p(C=k—x) are calculated and plotted in plots-2,3,4. p(x—C=k) is calculated using the normald function. p(x) is calculated by using the formula p(x—C=k)*p(Class=k). p(C=k—x) is calculated using the formula p(x)*p(x—C=k).
10. Lines[136-138]: Saving the plot with spacing adjustments between the sub-plots.

From the six different graphs shown in the figure 2 one can observe the following: The explanations for graph 1-4 are same as that of QDA. But, when we see the 5th plot, which is the discrimination graph, it can be observed to be straight lines. Hence, it is a linear boundary. but, there is no overlap of lines in this case. This explains the reason why the all the samples have been classified only to the 2nd class. The LDA method is not quite useful for the data set chosen. Classification is poor mainly because of the masking problem.

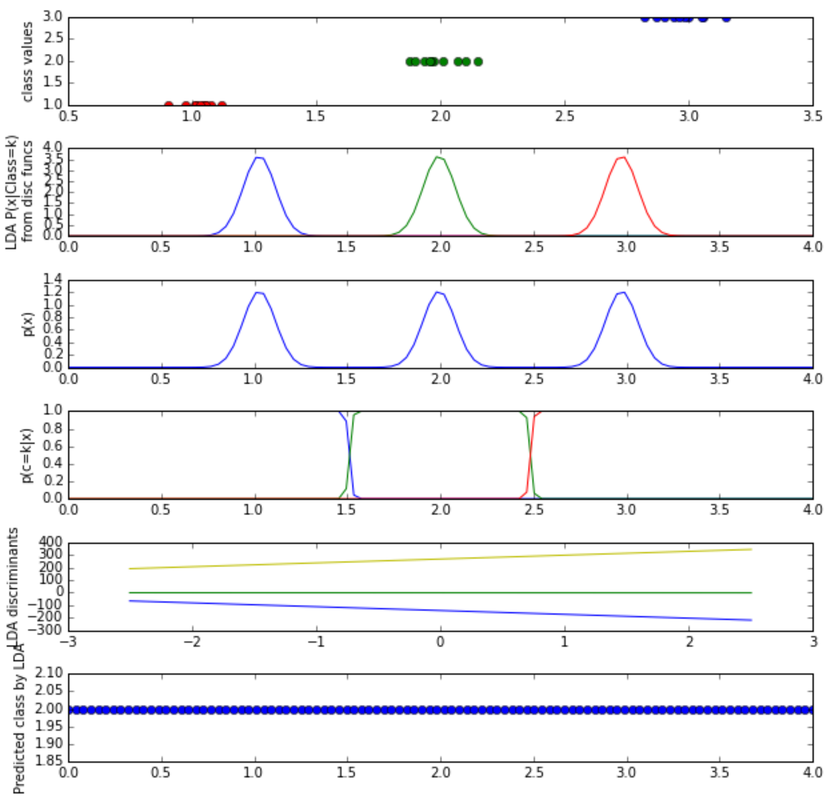

Figure 2: LDA-1D data

2.3 NLR method

The python code for Non-linear regression using neural network is shown below.

```python
import numpy as np
import matplotlib.pyplot as plt
#get neural network class
!rm nn1.tar.gz
!wget http://www.cs.colostate.edu/~anderson/cs545/notebooks/nn1.tar.gz
!tar xvf nn1.tar.gz
import neuralnetworks1 as nn

#training data creation
colors = ['blue','red','green']
D=1 #dimension = 1
N=10 #number of samples in each class
X1=np.random.normal(1.0,0.1,(N,D))
T1=np.array([1]*N).reshape((N,1))
X2=np.random.normal(2.0,0.1,(N,D))
T2=np.array([2]*N).reshape((N,1))
X3=np.random.normal(3.0,0.1,(N,D))
T3=np.array([3]*N).reshape((N,1))
data=np.hstack((np.vstack((X1,X2,X3)),np.vstack((T1,T2,T3))))
X=data[:,0:D]
T = np.repeat(range(1,4),n).reshape((-1,1))

#Neural network model bulding
nHidden = 5
nnet = nn.NeuralNetworkClassifier(X.shape[1],nHidden,3)
nnet.train(X,T, nIterations=1000)

#Test data creation
nNew = 100
Xtest = np.linspace(0,4,nNew).repeat(D).reshape((nNew,D))

#Predicting
Ytest = nnet.use(Xtest)
predTest,probs,_ = nnet.use(Xtest,allOutputs=True)  #discard hidden unit outputs

#Plots
plt.figure(figsize=(10,10))

#code for plotting between classes and data
plt.subplot(3,1,1)
plt.plot(X[class1rows],T[class1rows],"ro")
plt.plot(X[class2rows],T[class2rows],"go")
plt.plot(X[class3rows],T[class3rows],"bo")
plt.ylabel("class values")

#code for plotting the curve p(c=k|x)
plt.subplot(3,1,2)
plt.plot(Xtest,probs)
plt.ylabel("p(c=k|x)")

#code for plotting  the class predicted by the classifier for the test data.
plt.subplot(3,1,3)
preTest = np.argmax(probs,axis=1)
print(preTest.shape)
plt.plot(Xtest,preTest,'o')
plt.ylabel("Predicted class by NNC")
```

```
57
58 #code for saving the plot
59 plt.subplots_adjust(hspace=0.5,wspace = .5)
60 plt.savefig('nlrtoy.png')
```

The code has 7 parts.

1. Lines[1-7]: Normal initialization of packages. It imports the neural network class which has the main code for doing the classification.
2. Lines[9-21]:Training data creation similar as before.
3. Lines[23-26]: Training the neural network with the data created.
4. Lines[28-30]: Testing data creation of 100 samples.
5. Lines[32-34]: Prediction by using the testing data.
6. Lines[36-56]: Plots for plotting the 3 graphs. Plot 1 and 3 are nothing but Train and test data versus their corresponding classes. The 2nd plot is the p(C=k—x)plot for the 3 different classes whose formula is similar as before.
7. Lines[58-60]: Saving the plots and adjusting the sub-plots.

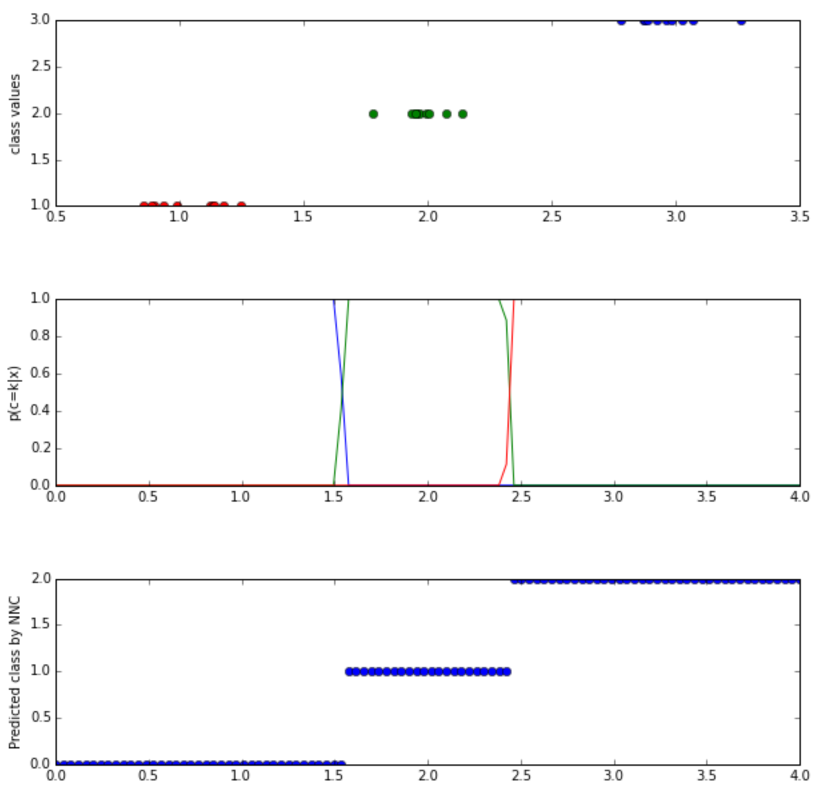

Figure 3: NLR-1D data

The 3 graphs are almost similar to that of the QDA analysis. Subplot 1 is the train data versus its class whereas 3rd sub-plot is that of the predicted classes for the test data.The 2nd sub-plot is the discriminant function and the final plot is the one that shows classes versus the test data.

2.4 LLR method

From the class notebook[3], I just introduced a negative sign in the objective function and the gradient. So, the formulas for the gradient and the update rule will be

$$-\sum_{n=1}^{N} x_n(t_{n,j} - g_j(x_n)) \tag{4}$$

$$\beta_j \leftarrow \beta_j - \alpha \sum_{n=1}^{N}(t_{n,j} - g_j(x_n))x_n \tag{5}$$

The python code is shown below.

```
import numpy as np
import matplotlib.pyplot as plt
import newnn as nn2
import mpl_toolkits.mplot3d as plt3
from matplotlib import cm
%matplotlib inline

#Building model
def makeLLR(X,T):
    nHidden = 1
    nnet1 = nn2.NeuralNetworkClassifier(X.shape[1],nHidden,len(np.unique(T)))
    nnet1.train(X, T,   nIterations=100)
    return nnet1

#Predicting
def useLLR(nnet1,X):
    predTest, probs, Z = nnet1.use(X,allOutputs=True)
    return predTest, probs, Z

#Test data creation
D = 1  # number of components in each sample
N = 10  # number of samples in each class
X1 = np.random.normal(1,0.1,(N,D))
T1 = np.array([1]*N).reshape((N,1))
X2 = np.random.normal(2,0.1,(N,D))  # wider variance
T2 = np.array([2]*N).reshape((N,1))
X3 = np.random.normal(3,0.1,(N,D))  # wider variance
T3 = np.array([3]*N).reshape((N,1))

data = np.hstack(( np.vstack((X1,X2,X3)), np.vstack((T1,T2,T3))))
X = data[:,0:D]
T = data[:,-1]
T = T.reshape(-1,1)

#Training data creation
nNew = 100
Xtest = np.linspace(0,4,nNew).repeat(D).reshape((nNew,D))

#Invoke the functions
model1 = makeLLR(X,T);
predTest,probs,Z = useLLR(model1,Xtest)
```

```

#Plots
plt.figure(figsize=(10,10))

#code for plotting between classes and data
class1rows=T==1
class2rows=T==2
class3rows=T==3
plt.subplot(3,1,1)
plt.plot(X[class1rows],T[class1rows],"ro")
plt.plot(X[class2rows],T[class2rows],"go")
plt.plot(X[class3rows],T[class3rows],"bo")
plt.ylabel("class values")

#code for plotting the curve p(c=k|x)
plt.subplot(3,1,2)
plt.plot(Xtest,probs)
plt.ylabel("p(c=k|x)")

#code for plotting  the class predicted by the classifier for the test data.
plt.subplot(3,1,3)
preTest = np.argmax(probs,axis=1)
plt.plot(Xtest,preTest,'o')
plt.ylabel("Predicted class by LLR")

#code for saving the plot
plt.subplots_adjust(hspace=0.5,wspace = .5)
plt.savefig('llrtoy.png')
```

The code is exactly same as that of the non-linear logistic except that I have used the makeLLR and useLLR function definitions(because this was the last thing I did in the report. So, included the latest version. The make*** and use** for QDA and LDA are included in Appendix A). The difference arises from the Neural network classifier code that is imported from newnn. The calculations for hidden layers have been removed and also the objective and gradient function has been changed. The difference has been captured using an online tool for identifying the difference between 2 texts [4]. The indicator variable definition has been introduced, the initial weights turned to be zero. The modified file is shown in Appendix B.

The graph from the above plotting codes is shown in figure 4. It is quite similar to that of the figure 3.

3 Real data set

Wine data set[2] from the UCI Machine learning repository is used because it has variety of things which can be a good example for classification. In the documentation, the prediction results of various classification has been provided. So, it is a good data set to test against the code defined by us and compare with the documented results as well. It has 13 attributes, the first column being the class identifier.The 13 attributes are Alcohol, Malic acid, Ash, Alcalininty of ash, Magnesium, Total phenols, Flavanoids, Nonflavnoid phenols, Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines and Proline. the python code for classifying this data set using the 4 classifiers is shown below.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
!rm nn1.tar.gz
!wget http://www.cs.colostate.edu/~anderson/cs545/notebooks/nn1.tar.gz
!tar xvf nn1.tar.gz
import neuralnetworks1 as nn
import newnn as nn2
import mpl_toolkits.mplot3d as plt3
from matplotlib import cm
```

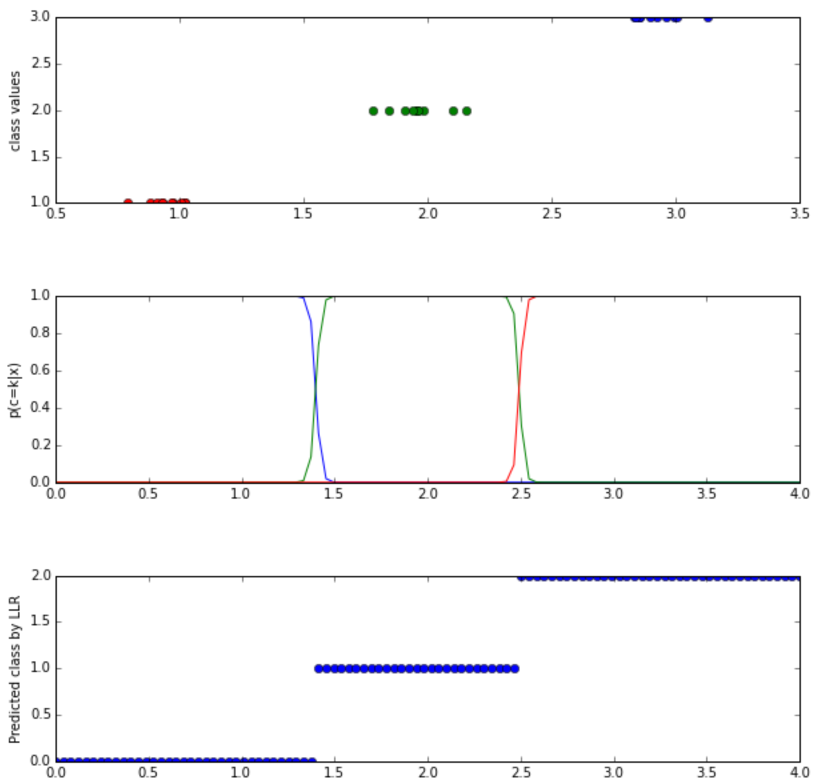

Figure 4: LLR-1D data

```python

#Standardization function
def makeStandardizeF(X):
    means = X.mean(axis=0)
    stds = X.std(axis=0,ddof=1)
    def standardize(origX):
        return (origX - means) / stds
    def unStandardize(stdX):
        return stds * stdX + means
    return (standardize, unStandardize)

#Random partitioning function
def ranPartition(X,T,Frac):
    (nrow, ncol)=X.shape
    nTrain=int(round(nrow*Frac))
    nTest=nrow-nTrain
    rows = np.arange(nrow)
    np.random.shuffle(rows)
    trainIndices = rows[:nTrain]
    testIndices = rows[nTrain:]
    Xtrain=X[trainIndices,:]
    Ttrain=T[trainIndices]
    Xtest=X[testIndices,:]
    Ttest=T[testIndices]
    return (Xtrain,Ttrain,Xtest,Ttest)

#QDA discrimination function
def discQDA(X, standardize, mu, Sigma, prior):
    Xc = standardize(X) - mu
    if Sigma.size == 1:
        Sigma = np.asarray(Sigma).reshape((1,1))
    det = np.linalg.det(Sigma)
    if det == 0:
        raise np.linalg.LinAlgError('discQDA(): Singular covariance matrix')
    SigmaInv = np.linalg.inv(Sigma)     # pinv in case Sigma is singular
    return -0.5 * np.log(det) \
           - 0.5 * np.sum(np.dot(Xc,SigmaInv) * Xc, axis=1) \
           + np.log(prior)

#LDA discrimination function
def discLDA(X,standardize,mu,Sigma,prior):
    Xc=standardize(X) #-mu
    if Sigma.size==1:
        Sigma=np.asarray(Sigma).reshape((1,1))
    det=np.linalg.det(Sigma)
    if det==0:
        raise np.linalg.LinAlgError('discLDA():Sigular covarience matrix')
    SigmaInv=np.linalg.inv(Sigma)
    return np.dot(X,np.dot(SigmaInv,mu))-0.5*np.dot
                          (np.dot(Xc,SigmaInv),mu)+np.log(prior)

#Building model and predicting functions for LLR
def makeLLR(X,T):
    nHidden = 1
    nnet1 = nn2.NeuralNetworkClassifier(X.shape[1],1,len(np.unique(T)))
    nnet1.train(X, T,  nIterations=100)
    return nnet1

def useLLR(nnet1,X):
```

```
    predTest, probs, Z = nnet1.use(X, allOutputs=True)
    return predTest, probs, Z

#Correctness measure function
def percentCorrect(p,t):
    return np.sum(p.ravel()==t.ravel()) / float(len(t)) * 100

#Loading the data
data = np.loadtxt("wine.data" ,delimiter=',')
X = data[:,1:]
T = data[:,0]

#Partitioning test and train and respective classes.
class1rows=T==1
class2rows=T==2
class3rows=T==3

X1train,T1train,X1test,T1test=ranPartition(X[class1rows,:],T[class1rows],0.8)
X2train,T2train,X2test,T2test=ranPartition(X[class2rows,:],T[class2rows],0.8)
X3train,T3train,X3test,T3test=ranPartition(X[class3rows,:],T[class3rows],0.8)

T1train=T1train.reshape(-1,1)
T2train=T2train.reshape(-1,1)
T3train=T3train.reshape(-1,1)


T1test=T1test.reshape(-1,1)
T2test=T2test.reshape(-1,1)
T3test=T3test.reshape(-1,1)

Xtrain=np.vstack((X1train,X2train,X3train))
Ttrain=np.vstack((T1train,T2train,T3train))

Xtest=np.vstack((X1test,X2test,X3test))
Ttest=np.vstack((T1test,T2test,T3test))

Ttrain1=Ttrain.ravel()
standardize,_ = makeStandardizeF(Xtrain)
Xtrains = standardize(Xtrain)

#Parameter definition
class1rows=Ttrain1==1
class2rows=Ttrain1==2
class3rows=Ttrain1==3

mu1=np.mean(Xtrains[class1rows,:],axis=0)
mu2=np.mean(Xtrains[class2rows,:],axis=0)
mu3=np.mean(Xtrains[class3rows,:],axis=0)

Sigma1=np.cov(Xtrains[class1rows,:].T)
Sigma2=np.cov(Xtrains[class2rows,:].T)
Sigma3=np.cov(Xtrains[class3rows,:].T)

Sigma=prior1 * Sigma1 + prior2 * Sigma2 + prior3 * Sigma3
#Sigma=(Sigma1+Sigma2+Sigma3)/3.0

N1=np.sum(class1rows)
N2=np.sum(class2rows)
N3=np.sum(class3rows)
```

```python

N=len(T)
prior1=N1/float(N)
prior2=N2/float(N)
prior3=N3/float(N)

nHidden = 1
nnet = nn.NeuralNetworkClassifier(X.shape[1],nHidden,3)
 # 3 classes, will actually make 2-unit output layer

#Model building and testing for NLR

nnet.train(Xtrain,Ttrain,nIterations=1000,errorPrecision=1.e-8)
predTest,probsTest,_ = nnet.use(Xtest,allOutputs=True)
#discard hidden unit outputs
predTrain,probsTrain,_ = nnet.use(Xtrain,allOutputs=True)

#Model building and testing for LLR

model1 = makeLLR(Xtrain,Ttrain);
predTestLLR,probsTestLLR,_ = useLLR(model1,Xtest)
predTrainLLR,probsTrainLLR,_ = useLLR(model1,Xtrain)

#Model building and testing for QDA

d1_train=discQDA(Xtrain,standardize,mu1,Sigma1,prior1)
d2_train=discQDA(Xtrain,standardize,mu2,Sigma2,prior2)
d3_train=discQDA(Xtrain,standardize,mu3,Sigma3,prior3)
predictedTrain=np.argmax(np.vstack((d1_train,d2_train,
d3_train)),axis=0).reshape(-1,1)

d1_test=discQDA(Xtest,standardize,mu1,Sigma1,prior1)
d2_test=discQDA(Xtest,standardize,mu2,Sigma2,prior2)
d3_test=discQDA(Xtest,standardize,mu3,Sigma3,prior3)
predictedTest=np.argmax(np.vstack((d1_test,d2_test,
d3_test)),axis=0).reshape(-1,1)


#Model building and testing for LDA
dl1_train=discLDA(Xtrain,standardize,mu1,Sigma,prior1)
dl2_train=discLDA(Xtrain,standardize,mu2,Sigma,prior2)
dl3_train=discLDA(Xtrain,standardize,mu3,Sigma,prior3)
predictedTrainLDA=np.argmax(np.vstack((dl1_train,dl2_train,
dl3_train)),axis=0).reshape(-1,1)


dl1_test=discLDA(Xtest,standardize,mu1,Sigma,prior1)
dl2_test=discLDA(Xtest,standardize,mu2,Sigma,prior2)
dl3_test=discLDA(Xtest,standardize,mu3,Sigma,prior3)
predictedTestLDA=np.argmax(np.vstack((dl1_test,dl2_test,dl3_test))
                                                            ,axis=0).reshape(-1,1)

#Correctness of the prediction
print("QDA Percent correct: Train",percentCorrect(predictedTrain+1,Ttrain),
          "Test",percentCorrect(predictedTest+1,Ttest))
print("LDA Percent correct: Train",percentCorrect(predictedTrainLDA+1,Ttrain),
          "Test",percentCorrect(predictedTestLDA+1,Ttest))
print("NLR Percent Correct: Train",percentCorrect(predTrain,Ttrain),
          "Test",percentCorrect(predTest,Ttest))
```

```
188 print("LLR Percent Correct: Train",percentCorrect(predTrainLLR,Ttrain),
189           "Test",percentCorrect(predTestLLR,Ttest))
```

The code has 13 parts.

1. Lines[1-10]: The initial package importing
2. Lines[12-20]: Standardization function definition as discussed before.
3. Lines[22-35]: Random data partitioning function which partitions the input and target variables into Training and testing data. So, a total of 4 outputs are expected from this function return.
4. Lines[37-48]: QDA discrimination function as discussed before.
5. Lines[50-60]: LDA discrimination function as discussed before.
6. Lines[62-71]: LLR make and use function as discussed before.
7. Lines[73-75]: Correctness measurement function which calculates the difference between the original and the predicted values and averages it. finally, gives the percentage of the result. It is nothing but the correctness percentage of the predicted value.
8. Lines[77-80]: Loading the wine.data into python.
9. Lines[82-108]: Calling the partition functions with respect to each class and finally concatenating them.
10. Lines[110-137]: Parameters for the models are been found namely mu, sigma and prior probabilities. The neural network hidden layers is fixed as 1 and an object to the class is been created.
11. Lines[139-145]: Model building and testing for NLR classification as discussed previously.
12. Lines[146-150]: Model building and testing for LLR classification as discussed previously.
13. Lines[152-164]: Model building and testing for QDA classification as discussed previously. It is performed on each class for both test and training data. The predicted results are concatenated.
14. Lines[167-179]: Model building and testing for LDA classification as discussed previously. It is performed on each class for both test and training data. The predicted results are concatenated.
15. Lines[181-189]: Calls the correctness measure function for all the models and prints the results.

The results obtained for the correctness measure is shown below. It can be seen that Neural network and LLR performs well. There were instances where QDA showed 100 percent accuracy on the Training data. But, when comparing the testing data, QDA seems to outperform others.

```
QDA Percent correct: Train 99.2957746479 Test 100.0
LDA Percent correct: Train 33.0985915493 Test 33.3333333333
NLR Percent Correct: Train 100.0 Test 94.4444444444
LLR Percent Correct: Train 100.0 Test 98.0083678988
```

4 Additional question

I was excited about the 3D plots in the 17th notebook. So, wanted to know the behaviour of any one of the classification methods(say, QDA) on 2D data and visualize the plots. The following gives the code implemented.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

#Training data creation
D=2 #number of components in each sample or dimension
N=10 #number of samples in each class
X1=np.random.normal(1.0,0.1,(N,D))
T1=np.array([1]*N).reshape((N,1))
X2=np.random.normal(2.0,0.1,(N,D))
T2=np.array([2]*N).reshape((N,1))
X3=np.random.normal(3.0,0.1,(N,D))
T3=np.array([3]*N).reshape((N,1))
data=np.hstack((np.vstack((X1,X2,X3)),np.vstack((T1,T2,T3))))
X=data[:,0:D]
T=data[:,-1]
standardize,_=makeStandardizeF(X)
Xs=standardize(X)

#Parameter creation
class1rows=T==1
class2rows=T==2
class3rows=T==3

mu1=np.mean(Xs[class1rows,:],axis=0)
mu2=np.mean(Xs[class2rows,:],axis=0)
mu3=np.mean(Xs[class3rows,:],axis=0)

Sigma1=np.cov(Xs[class1rows,:].T)
Sigma2=np.cov(Xs[class2rows,:].T)
Sigma3=np.cov(Xs[class3rows,:].T)

N1=np.sum(class1rows)
N2=np.sum(class2rows)
N3=np.sum(class3rows)

N=len(T)
prior1=N1/float(N)
prior2=N2/float(N)
prior3=N3/float(N)

#Model building
d1=discQDA(newData,standardize,mu1,Sigma1,prior1)
d2=discQDA(newData,standardize,mu2,Sigma2,prior2)
d3=discQDA(newData,standardize,mu3,Sigma3,prior3)
colors=['blue','red','green']

#Plotting codes
figure=plt.figure(1)
plt.clf()
classes=[1,2,3]
axes=Axes3D(figure)
for i in range(3):
    r=(T==classes[i]).flatten()
    Xi=X[r,:]
    axes.scatter(Xi[:,0].ravel(),Xi[:,1].ravel(),T[r].ravel(),'o',color=colors[i])
axes.set_zlabel('Train class')
```

```

#code for plotting  the three curves for p(x|C=k)
for k=1,2,3, for x values in a set of test data generated
by x = np.linspace(0,4,100), where p(x|C=k) is calculated
using means and standard deviations for each class
calculated from the training data,
#plt.subplot(6,1,2)

p1= normald(newDataS,mu1,Sigma1)
p2= normald(newDataS,mu2,Sigma2)
p3= normald(newDataS,mu3,Sigma3)
p1.resize(a.shape)
p2.resize(a.shape)
p3.resize(a.shape)
px=p1*prior1+p2*prior2+p3*prior3
px.resize(a.shape)
add=1
pofc1=p1*prior1/px
pofc2=p2*prior2/px
pofc3=p3*prior3/px
pofc1.resize(a.shape)
pofc2.resize(a.shape)
pofc3.resize(a.shape)
d1.resize(a.shape)
d2.resize(a.shape)
d3.resize(a.shape)

figure_2=plt.figure()
axes=Axes3D(figure_2)
axes.plot_surface(a,b,p1,rstride=1,cstride=1,cmap=cm.jet)
axes.set_title('p(x|c=k) for 2D')

#figure_3=plt.figure()
#axes=Axes3D(figure_3)
axes.plot_surface(a,b,p2,rstride=1,cstride=1,cmap=cm.jet)
#axes.set_title('p(x|c=2) for 2D')

#figure_4=plt.figure()
#axes=Axes3D(figure_4)
axes.plot_surface(a,b,p3,rstride=1,cstride=1,cmap=cm.jet)
#axes.set_title('p(x|c=3) for 3D')

#code for plotting the curve p(x) for the test data......
figure_5=plt.figure()
axes=Axes3D(figure_5)
axes.plot_surface(a,b,px,rstride=1,cstride=1,cmap=cm.jet)
axes.set_title('p(x) for the test data')

#code for plotting the curve p(c=k|x)
figure_6=plt.figure()
axes=Axes3D(figure_6)
axes.plot_surface(a,b,pofc1,rstride=1,cstride=1,cmap=cm.jet)
axes.set_title('QDA p(c=1|x) for 2D')
figure_7=plt.figure()
axes=Axes3D(figure_7)
axes.plot_surface(a,b,pofc2,rstride=1,cstride=1,cmap=cm.jet)
axes.set_title('QDA p(c=2|x) for 2D')
figure_8=plt.figure()
axes=Axes3D(figure_8)
```

```
axes.plot_surface(a,b,pofc3,rstride=1,cstride=1,cmap=cm.jet)
axes.set_title('QDA p(c=3|x) for 2D')

#code for plotting the discriminants
dq=np.vstack((d1,d2,d3))
figure_9=plt.figure()
axes=Axes3D(figure_9)
axes.plot_surface(a,b,d1,rstride=1,cstride=1,color='red')
axes.plot_surface(a,b,d2,rstride=1,cstride=1,color='green')
axes.plot_surface(a,b,d3,rstride=1,cstride=1,color='blue')
axes.set_title('QDA discriminant for 2D')

disc = np.hstack((d1.reshape(-1,1),d2.reshape(-1,1),d3.reshape(-1,1)))
preTest = np.argmax(disc, axis=1)
#preTest = np.argmax(np.vstack((d1,d2,d3)), axis=1)
plt.figure(10)
#ax = Axes3D(fig10)
#ax.scatter(newData,(preTest+1), 'o', color = 'green')
#ax.set_title('QDA Class Prediction for testing data')
classes=[1,2,3]
plt.plot(newData,(preTest+1),'o')
#[:,0].reshape((100,100)),newData[:,1].reshape((100,100))
```

The code has the similar structure as explained previously for one dimensional data. The same list of plotting codes are used except that they are now in 3D space and so the Axes3D package was used for this purpose. The graphs plotted by the above code is shown in figures 5-12. Figure 5 shows the classes versus 2D data. From the figure 6 one can observe the probability p(x|c=k) is high for two classes when compared to the third one. Figure 7 shows the total probability p(x). From figure 8,9,10 one can observe that the probability(p(c=k|x)) is plotted which is calculated as p(x|c=k)/p(x). From the figure 11 one can clearly observe that discriminants are slightly curved this is because they are not calculated linearly. From the figure 12 one can observe predicted classes for the test data. All the figures are included in Appendix C.

5 Conclusion

Thus, the classification methods have been studied for the 4 algorithms as mentioned above. It can be concluded that when the precision of the decision boundary is more accurate, we get a good classification. This is evident from the masking effect of LDA which is further improved by other methods. The implementation has been well demonstrated in the one-dimensional data and the accuracy has been studied well in the real data sample.

References

[1] Christopher M.Bishop., "Pattern Recognition and Machine Learning", Springer, 2006.

[2] `https://archive.ics.uci.edu/ml/datasets/Wine`

[3] `http://nbviewer.ipython.org/url/www.cs.colostate.edu/~anderson/cs545/notebooks/15%20Classification%20with%20Linear%20Logistic%20Regression.ipynb`

[4] `https://www.diffchecker.com/diff`

A Appendix A: Formatted codes

For the experiment purposes, I used the code presented in the report because, the formatted (make*** and use***) was a bit time consuming to create and validate. I made a little effort to re-write it in the formatted form after completing the report. Here are the codes. the following is the code for LDA. It is the same for QDA except that SigmaInv has to be calculated for each Sigma value.

```
def makeLDA(Xtrain, Ttrain):
    standardize, _ = makeStandardize(X)
    Xs = standardize(X)
    class1rows = Ttrain==1
    class2rows = Ttrain==2
    class3rows = Ttrain==3

    mu1 = np.mean(Xs[class1rows,:], axis=0)
    mu2 = np.mean(Xs[class2rows,:], axis=0)
    mu3 = np.mean(Xs[class3rows,:], axis=0)

    Sigma1 = np.cov(Xs[class1rows,:].T)
    Sigma2 = np.cov(Xs[class2rows,:].T)
    Sigma3 = np.cov(Xs[class3rows,:].T)

    N1 = np.sum(class1rows)
    N2 = np.sum(class2rows)
    N3 = np.sum(class3rows)

    N = len(Ttrain)
    prior1 = N1 / float(N)
    prior2 = N2 / float(N)
    prior3 = N3 / float(N)

    SIGMA = (prior1*Sigma1) + (prior2*Sigma2) + (prior3*Sigma3)

    return (mu1,mu2,mu3, Sigma1,Sigma2,Sigma3,prior1, prior2, prior3,SIGMA)

def useLDA(model,X):
    mu1,mu2,mu3, Sigma1,Sigma2,Sigma3,prior1, prior2, prior3,SIGMA = model
    standardize, _ = makeStandardize(X)
    Xc1 = standardize(X)
    Xc2 = standardize(X)
    Xc3 = standardize(X)

    if SIGMA.size == 1:
        SIGMA = np.asarray(SIGMA).reshape((1,1))
    det = np.linalg.det(SIGMA)
    if det == 0:
        raise np.linalg.LinAlgError('discLDA(): Singular covariance matrix')
    SigmaInv = np.linalg.inv(SIGMA)    # pinv in case Sigma is singular

    d1 = np.dot(np.dot(SigmaInv, mu1), Xc1.T) -
    (0.5 * np.dot(np.dot(mu1.T, SigmaInv), mu1)) + np.log(prior1)
    d2 = np.dot(np.dot(SigmaInv, mu2), Xc2.T) -
    (0.5 * np.dot(np.dot(mu2.T, SigmaInv), mu2)) + np.log(prior2)
    d3 = np.dot(np.dot(SigmaInv, mu3), Xc3.T) -
    (0.5 * np.dot(np.dot(mu3.T, SigmaInv), mu3)) + np.log(prior3)

    probs = np.exp( np.vstack((d1,d2,d3)).T -
    0.5*D*np.log(2*np.pi) - np.log(np.array([[prior1,prior2,prior3]])))
    predictedTest = np.argmax(np.vstack((d1,d2,d3)),axis=0)
    newDataS = standardize(newData)
    probs_normald = np.hstack((normald(newDataS,mu1,Sigma1),normald(newDataS,mu2,
    Sigma2),normald(newDataS,mu3,Sigma3)))

```

```
    return predictedTest, probs,probs_normald, d1,d2,d3
```

B Appendix B: Modified Neural network file

The modified is shown. I thought of including the result from the difference finder [4]. But, didn't have sufficient time to find means to include in Latex.

```
''' Neural network with one hidden layer.
 For nonlinear regression (prediction of real-valued outputs)
   net = NeuralNetwork(ni,nh,no)       # ni is number of attributes each sample,
                                       # nh is number of hidden units,
                                       # no is number of output components
   net.train(X,T,                      # X is nSamples x ni, T is nSamples x no
             nIterations=1000,      # maximum number of SCG iterations
             weightPrecision=1e-8, # SCG terminates
              when weight change magnitudes fall below this,
             errorPrecision=1e-8)  # SCG terminates when objective
              function change magnitudes fall below this
   Y,Z = net.use(Xtest,allOutputs=True)  # Y is nSamples x no, Z is
   nSamples x nh

 For nonlinear classification (prediction of integer valued class labels)
   net = NeuralNetworkClassifier(ni,nh,no)
   net.train(X,T,                      # X is nSamples x ni, T is
   nSamples x 1 (integer class labels
             nIterations=1000,      # maximum number of SCG iterations
             weightPrecision=1e-8, # SCG terminates when weight
             change magnitudes fall below this,
             errorPrecision=1e-8)  # SCG terminates when objective
              function change magnitudes fall below this
   classes,Y,Z = net.use(Xtest,allOutputs=True)  # classes is nSamples x 1
'''



import scaledconjugategradient as scg
# reload(gd)
import numpy as np
import matplotlib.pyplot as plt
import multiprocessing as mp  # to allow access to number of elapsed iterations
#import cPickle
import mlutils as ml

# def pickleLoad(filename):
#     with open(filename,'rb') as fp:
#         nnet = cPickle.load(fp)
#     nnet.iteration = mp.Value('i',0)
#     nnet.trained = mp.Value('b',False)
#     return nnet

class NeuralNetwork:
    def __init__(self,ni,nhs,no):
        try:
            nihs = [ni] + list(nhs)
        except:
            nihs = [ni] + [nhs]
            nhs = [nhs]
        self.Vs = [np.random.uniform(-0.1,0.1,size=(1+nihs[i],nihs[i+1]))
         for i in range(len(nihs)-1)]
```

```
        self.W = np.random.uniform(0,0,size=(1+nhs[-1],no))
        # print [v.shape for v in self.Vs], self.W.shape
        self.ni, self.nhs, self.no = ni, nhs, no
        self.Xmeans = None
        self.Xstds = None
        self.Tmeans = None
        self.Tstds = None
        self.iteration = mp.Value('i',0)
        self.trained = mp.Value('b',False)
        self.reason = None
        self.errorTrace = None

    def getSize(self):
        return (self.ni, self.nhs, self.no)

    def getErrorTrace(self):
        return self.errorTrace

    def getNumberOfIterations(self):
        return self.numberOfIterations

    def train(self,X,T,
              nIterations=100,weightPrecision=0,errorPrecision=0,verbose=False):
        if self.Xmeans is None:
            self.Xmeans = X.mean(axis=0)
            self.Xstds = X.std(axis=0)
        X = self._standardizeX(X)

        if T.ndim == 1:
            T = T.reshape((-1,1))

        if self.Tmeans is None:
            self.Tmeans = T.mean(axis=0)
            self.Tstds = T.std(axis=0)
        T = self._standardizeT(T)
        # Local functions used by gradientDescent.scg()

        def makeIndicatorVars(T):
            # Make sure T is two-dimensiona. Should be nSamples x 1.
            if T.ndim == 1:
                T = T.reshape((-1,1))
        return (T == np.unique(T)).astype(int)

        def objectiveF(w):
            self._unpack(w)
            Y,_ = self._forward_pass(X)
            return 0.5 * np.mean((Y - T)**2)

        def gradF(w):
            self._unpack(w)
            Y,Z = self._forward_pass(X)
            delta = (Y - T) / (X.shape[0] * T.shape[1])
            dVs,dW = self._backward_pass(delta,Z)
            return self._pack(dVs,dW)

        scgresult = scg.scg(self._pack(self.Vs,self.W), objectiveF, gradF,
                            xPrecision = weightPrecision,
                            fPrecision = errorPrecision,
                            nIterations = nIterations,
```

```
                              iterationVariable = self.iteration,
                              ftracep=True,
                              verbose=verbose)

        self._unpack(scgresult['x'])
        self.reason = scgresult['reason']
        self.errorTrace = scgresult['ftrace']
        self.numberOfIterations = len(self.errorTrace) - 1
        self.trained.value = True
        return self

    def use(self,X, allOutputs=False):
        Xst = self._standardizeX(X)
        Y,Z = self._forward_pass(Xst)
        Y = self._unstandardizeT(Y)
        return (Y,Z[1:]) if allOutputs else Y

    def draw(self,inputNames = None, outputNames = None):
        ml.draw(self.Vs, self.W, inputNames, outputNames)

    def __repr__(self):
        str = 'NeuralNetwork({}, {}, {})'.format(self.ni,self.nhs,self.no)
        # str += '  Standardization parameters' + (' not' if self.Xmeans ==
        None else '') + ' calculated.'
        if self.trained:
            str += '\n    Network was trained for {} iterations.
            Final error is {}.'.format(self.numberOfIterations,
            self.errorTrace[-1])
        else:
            str += '  Network is not trained.'
        return str

    def _standardizeX(self,X):
        return (X - self.Xmeans) / self.Xstds
    def _unstandardizeX(self,Xs):
        return self.Xstds * Xs + self.Xmeans
    def _standardizeT(self,T):
        return (T - self.Tmeans) / self.Tstds
    def _unstandardizeT(self,Ts):
        return self.Tstds * Ts + self.Tmeans

    def _forward_pass(self,X):
        Zprev = X
        Zs = [Zprev]
        """
        for i in range(len(self.nhs)):
            V = self.Vs[i]
            Zprev = np.tanh(np.dot(Zprev,V[1:,:]) + V[0:1,:])
            Zs.append(Zprev)
        """
        Y = np.dot(Zprev, self.W[1:,:]) + self.W[0:1,:]
        return Y, Zs

    def _backward_pass(self,delta,Z):
        dW = np.vstack((np.dot(np.ones((1,delta.shape[0])),delta),
         np.dot( Z[-1].T, delta)))
        dVs = []
        #delta = (1-Z[-1]**2) * np.dot( delta, self.W[1:,:].T)
        """
```

```
        for Zi in range(len(self.nhs),0,-1):
            Vi = Zi - 1 # because X is first element of Z
            dV = np.vstack(( np.dot(np.ones((1,delta.shape[0])), delta),
                             np.dot( Z[Zi-1].T, delta)))
            dVs.insert(0,dV)
            delta = np.dot( delta, self.Vs[Vi][1:,:].T) * (1-Z[Zi-1]**2)
        """
        return dVs,dW

    def _pack(self,Vs,W):
        # r = np.hstack([V.flat for V in Vs] + [W.flat])
        # print 'pack',len(Vs), Vs[0].shape, W.shape,r.shape
        # return np.hstack([V.flat for V in Vs] + [W.flat])
        return np.hstack([W.flat])

    def _unpack(self,w):
        first = 0
        numInThisLayer = self.ni
        """
        for i in range(len(self.Vs)):
            self.Vs[i][:] = w[first:first+(numInThisLayer+1)*self.nhs[i]].reshape(
                            (numInThisLayer+1,self.nhs[i]))
            first += (numInThisLayer+1) * self.nhs[i]
            numInThisLayer = self.nhs[i]
        """
        self.W[:] = w[first:].reshape((numInThisLayer+1,self.no))

    def pickleDump(self,filename):
        # remove shared memory objects. Can't be pickled
        n = self.iteration.value
        t = self.trained.value
        self.iteration = None
        self.trained = None
        with open(filename,'wb') as fp:
        #     pickle.dump(self,fp)
            cPickle.dump(self,fp)
        self.iteration = mp.Value('i',n)
        self.trained = mp.Value('b',t)

class NeuralNetworkClassifier(NeuralNetwork):
    def __init__(self,ni,nhs,no):
        #super(NeuralNetworkClassifier,self).__init__(ni,nh,no)
        NeuralNetwork.__init__(self,ni,nhs,no-1)

    def _multinomialize(self,Y):
        # fix to avoid overflow
        mx = np.max(Y)
        expY = np.exp(Y-mx)
        denom = np.exp(-mx) + np.sum(expY,axis=1).reshape((-1,1))
        # Y = np.hstack((expY / denom, 1/denom))
        rowsHavingZeroDenom = denom == 0.0
        if np.sum(rowsHavingZeroDenom) > 0:
            Yshape = (expY.shape[0],expY.shape[1]+1)
            nClasses = Yshape[1]
            # add random values to result in random choice of class
            Y = np.ones(Yshape) * 1.0/nClasses + np.random.uniform(0,0.1,Yshape)
            Y /= np.sum(Y,1).reshape((-1,1))
        else:
            Y = np.hstack((expY / denom, np.exp(-mx)/denom))
```

```
            return Y

    def train(self,X,T,
                  nIterations=100,weightPrecision=0,error
                               Precision=0,verbose=False):
        if self.Xmeans is None:
            self.Xmeans = X.mean(axis=0)
            self.Xstds = X.std(axis=0)
        X = self._standardizeX(X)

        self.classes = np.unique(T)
        if self.no != len(self.classes)-1:
            raise ValueError(" In NeuralNetworkClassifier, the number
             of outputs must be one less than\n the number of classes
             in the training data. The given number of outputs\n is %d
             and number of classes is %d. Try changing the number of
             outputs in the\n call to NeuralNetworkClassifier()." %
             (self.no,  len(self.classes)))
        T = ml.makeIndicatorVars(T)
        beta = np.zeros((X.shape[1],T.shape[1]-1))
        alpha = 0.0001

        def g(self,X,beta):
            fs = np.exp(np.dot(X, beta))  # N x K-1
            denom = (1 + np.sum(fs,axis=1)).reshape((-1,1))
            gs = - fs / denom
            return np.hstack((gs,1/denom))
        # Local functions used by gradientDescent.scg()
        def objectiveF(w):
            self._unpack(w)
            Y,_ = self._forward_pass(X)
            Y = self._multinomialize(Y)
            #gs = g(X,beta)
            return -np.sum(T * np.log(Y))/X.shape[0]

        def gradF(w):
            self._unpack(w)
            Y,Z = self._forward_pass(X)
            Y = self._multinomialize(Y)
            delta =  (Y[:,:-1] - T[:,:-1]) / (X.shape[0] * (T.shape[1]-1))
                  # alpha * np.dot(X.T, T[:,:-1] - Y[:,:-1])

            dVs,dW = self._backward_pass(delta,Z)
            return self._pack(dVs,dW)

        scgresult = scg.scg(self._pack(self.Vs,self.W), objectiveF, gradF,
                            xPrecision = weightPrecision,
                            fPrecision = errorPrecision,
                            nIterations = nIterations,
                            iterationVariable = self.iteration,
                            ftracep=True, verbose=verbose)

        self._unpack(scgresult['x'])
        self.reason = scgresult['reason']
        self.errorTrace = scgresult['ftrace']
        self.numberOfIterations = len(self.errorTrace) - 1
        self.trained.value = True
        return self

```

```
    def use(self,X, allOutputs=False):
        Xst = self._standardizeX(X)
        Y,Z = self._forward_pass(Xst)
        Y = self._multinomialize(Y)
        classes = self.classes[np.argmax(Y, axis=1)].reshape((-1,1))
        return (classes,Y,Z[1:]) if allOutputs else classes


if __name__== "__main__":
    plt.ion()

    print( '\n------------------------------------------------------
                                              -----------')
    print( "Regression Example: Approximate f(x) = 1.5 + 0.6 x
                                              + 0.4 sin(x)")
    # print( '                          Neural net with 1 input, 5 hidden
                                units, 1 output')
    nSamples = 10
    X = np.linspace(0,10,nSamples).reshape((-1,1))
    T = 1.5 + 0.6 * X + 0.8 * np.sin(1.5*X)
    T[np.logical_and(X > 2, X < 3)] *= 3
    T[np.logical_and(X > 5, X < 7)] *= 3

    nSamples = 100
    Xtest = np.linspace(0,10,nSamples).reshape((-1,1)) + 10.0/nSamples/2
    Ttest = 1.5 + 0.6 * Xtest + 0.8 * np.sin(1.5*Xtest)
     + np.random.uniform(-2,2,size=(nSamples,1))
    Ttest[np.logical_and(Xtest > 2, Xtest < 3)] *= 3
    Ttest[np.logical_and(Xtest > 5,Xtest < 7)] *= 3

    # # nnet = NeuralNetwork(1,(5,4,3,2),1)
    # # nnet = NeuralNetwork(1,(10,2,10),1)
    # # nnet = NeuralNetwork(1,(5,5),1)
    nnet = NeuralNetwork(1,(3,3,3,3),1)

    nnet.train(X,T, errorPrecision=1.e-10, weightPrecision=1.e-10
                                              , nIterations=1000)
    print( "scg stopped after",nnet.getNumberOfIterations(),
                                "iterations:",nnet.reason)
    Y = nnet.use(X)
    Ytest, Ztest = nnet.use(Xtest, allOutputs=True)
    print( "Final RMSE: train", np.sqrt(np.mean((Y-T)**2)),"test",
                          np.sqrt(np.mean((Ytest-Ttest)**2)))

    # import time
    # t0 = time.time()
    # for i in range(100000):
    #     Ytest,Ztest = nnet.use(Xtest, allOutputs=True)
    # print( 'total time to make 100000 predictions:',time.time() - t0)

    # print( 'Inputs, Targets, Estimated Targets')
    # print( np.hstack((X,T,Y)))

    plt.figure(1)
    plt.clf()

    nHLayers = len(nnet.nhs)
    nPlotRows = 3 + nHLayers

```

```
    plt.subplot(nPlotRows,2,1)
    plt.plot(nnet.getErrorTrace())
    plt.xlabel('Iterations');
    plt.ylabel('RMSE')

    plt.title('Regression Example')
    plt.subplot(nPlotRows,2,3)
    plt.plot(X,T, 'o-')
    plt.plot(X,Y, 'o-')
    plt.text(8,12, 'Layer {}'.format(nHLayers+1))
    plt.legend(('Train Target','Train NN Output'),loc='lower right',
               prop={'size':9})
    plt.subplot(nPlotRows,2,5)
    plt.plot(Xtest,Ttest,'o-')
    plt.plot(Xtest,Ytest,'o-')
    plt.xlim(0,10)
    plt.text(8,12, 'Layer {}'.format(nHLayers+1))
    plt.legend(('Test Target','Test NN Output'),loc='lower right',
               prop={'size':9})
    colors = ('blue','green','red','black','cyan','orange')
    for i in range(nHLayers):
        layer = nHLayers-i-1
        plt.subplot(nPlotRows,2,i*2+7)
        plt.plot(Xtest,Ztest[layer]) #,color=colors[i])
        plt.xlim(0,10)
        plt.ylim(-1.1,1.1)
        plt.ylabel('Hidden Units')
        plt.text(8,0, 'Layer {}'.format(layer+1))

    plt.subplot(2,2,2)
    nnet.draw(['x'],['sine'])
    plt.draw()


    # Now train multiple nets to compare error for different
                                  numbers of hidden layers

    if False:  # make True to run multiple network experiment

        def experiment(hs,nReps,nIter,X,T,Xtest,Ytest):
            results = []
            for i in range(nReps):
                nnet = NeuralNetwork(1,hs,1)
                nnet.train(X,T,weightPrecision=0,errorPrecision=0,
                                            nIterations=nIter)

                (Y,Z) = nnet.use(X, allOutputs=True)
                Ytest = nnet.use(Xtest)
                rmsetrain = np.sqrt(np.mean((Y-T)**2))
                rmsetest = np.sqrt(np.mean((Ytest-Ttest)**2))
                results.append([rmsetrain,rmsetest])
            return results

        plt.figure(2)
        plt.clf()

        results = []
        # hiddens [ [5]*i for i in range(1,6) ]
        hiddens = [[12], [6,6], [4,4,4], [3,3,3,3], [2,2,2,2,2,2],
```

```
                    [24], [12]*2, [8]*3, [6]*4, [4]*6, [3]*8, [2]*12]

        for hs in hiddens:
            r = experiment(hs,30,100,X,T,Xtest,Ttest)
            r = np.array(r)
            means = np.mean(r,axis=0)
            stds = np.std(r,axis=0)
            results.append([hs,means,stds])
            print( hs, means,stds)
        rmseMeans = np.array([x[1].tolist() for x in results])
        plt.clf()
        plt.plot(rmseMeans, 'o-')
        ax = plt.gca()
        plt.xticks(range(len(hiddens)),[str(h) for h in hiddens])
        plt.setp(plt.xticks()[1], rotation=30)
        plt.ylabel('Mean RMSE')
        plt.xlabel('Network Architecture')


    print( '\n--------------------------------------------------------------
                                                   ----------------')
    print( "Classification Example: XOR, approximate f(x1,x2)
                                                          = x1 xor x2")
    print( '                              Using neural net with 2
    inputs, 3 hidden units, 2 outputs')
    X = np.array([[0,0],[1,0],[0,1],[1,1]])
    T = np.array([[1],[2],[2],[1]])
    nnet = NeuralNetworkClassifier(2,(4,),2)
    nnet.train(X,T,weightPrecision=1.e-10,errorPreci
                              sion=1.e-10,nIterations=100)
    print( "scg stopped after",nnet.getNumberOfIterations()
                                    ,"iterations:",nnet.reason)
    (classes,y,Z) = nnet.use(X, allOutputs=True)


    print( 'X(x1,x2), Target Classses, Predicted Classes')
    print( np.hstack((X,T,classes)))

    print( "Hidden Outputs")
    print( Z)

    plt.figure(3)
    plt.clf()
    plt.subplot(2,1,1)
    plt.plot(np.exp(-nnet.getErrorTrace()))
    plt.xlabel('Iterations');
    plt.ylabel('Likelihood')
    plt.title('Classification Example')
    plt.subplot(2,1,2)
    nnet.draw(['x1','x2'],['xor'])
```

C Appendix C: Figures of additional question

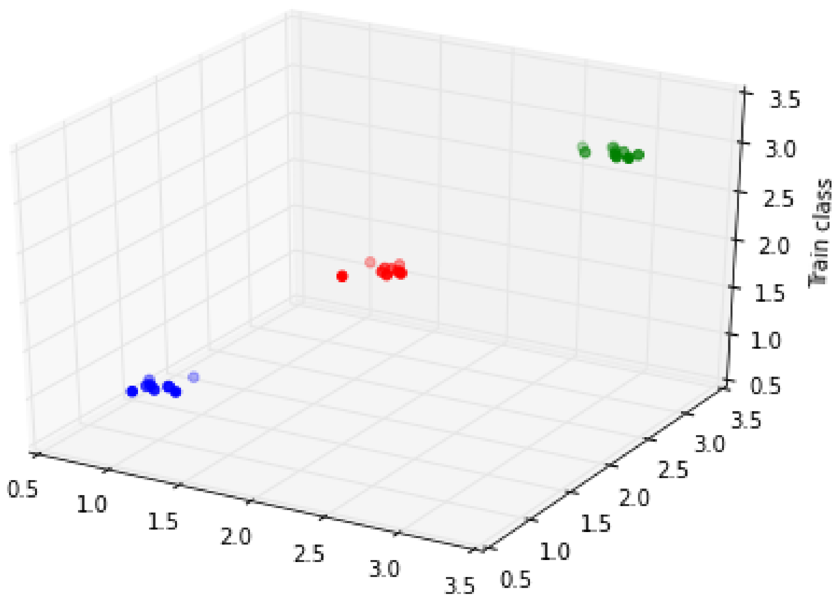

Figure 5: Plot

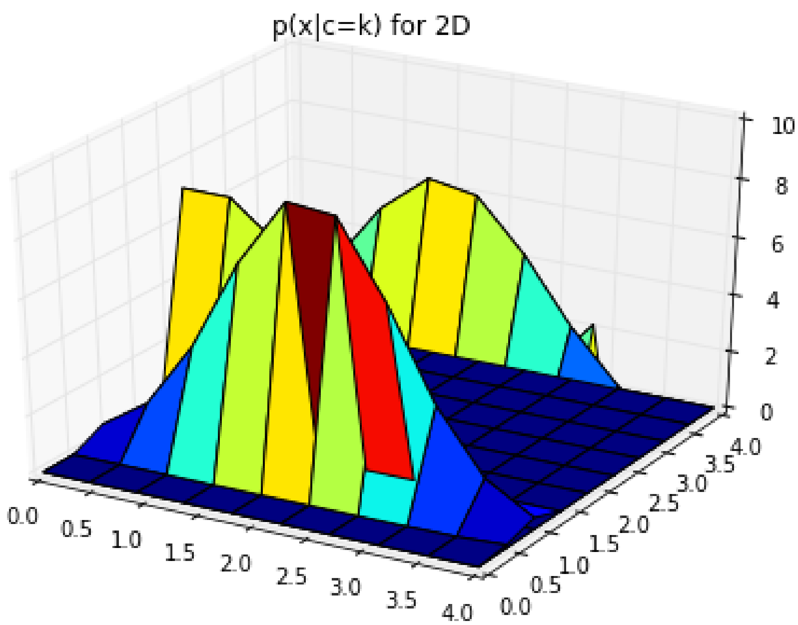

Figure 6: Plot

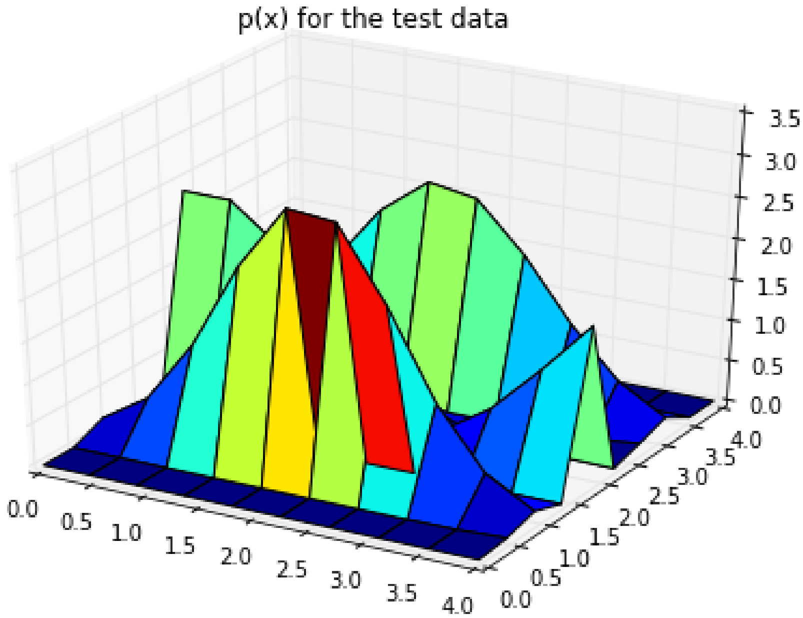

Figure 7: Plot

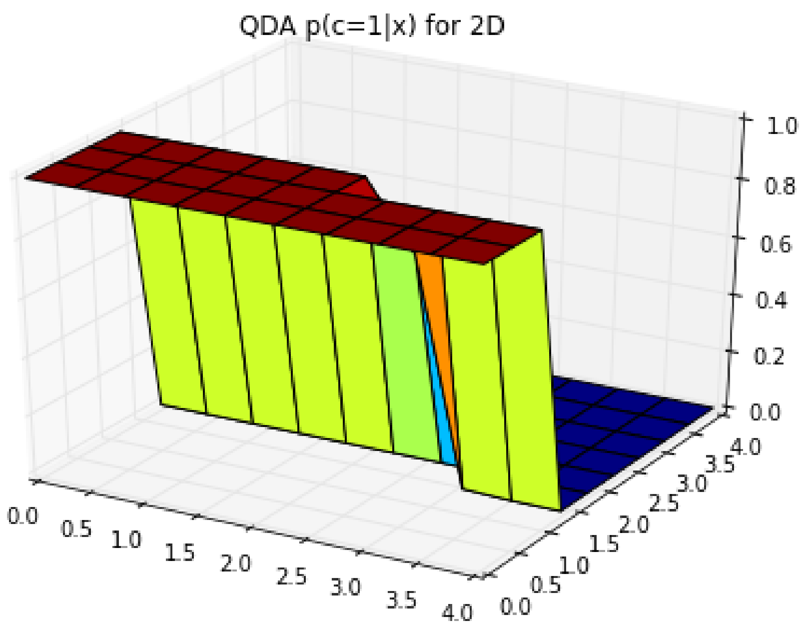

Figure 8: Plot

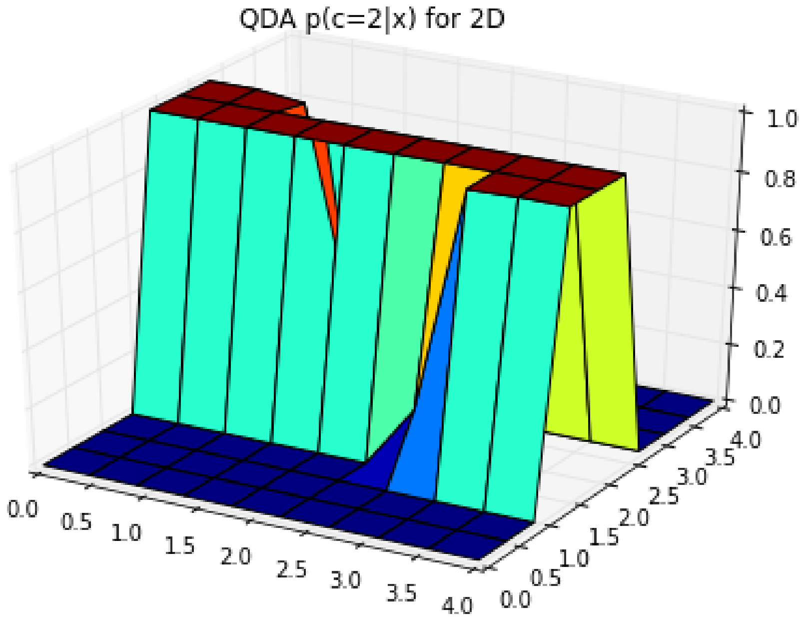

Figure 9: Plot

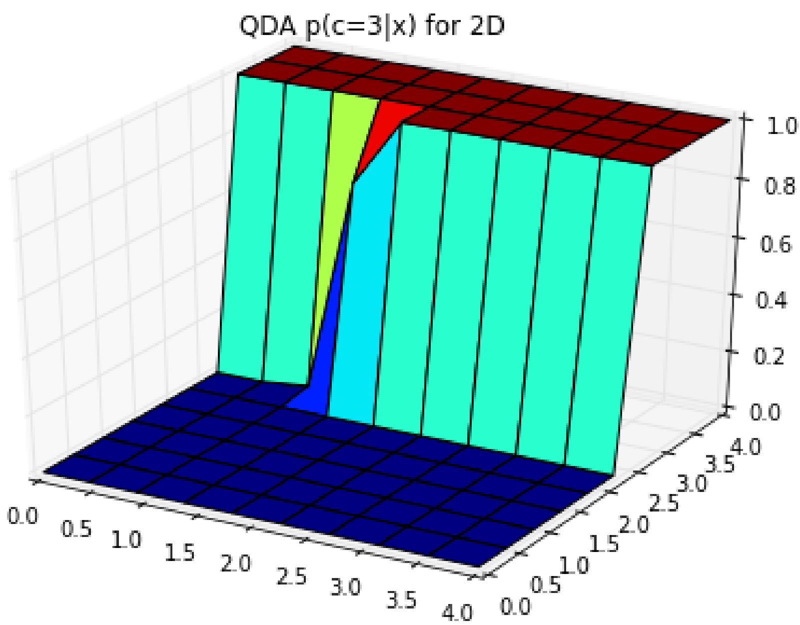

Figure 10: Plot

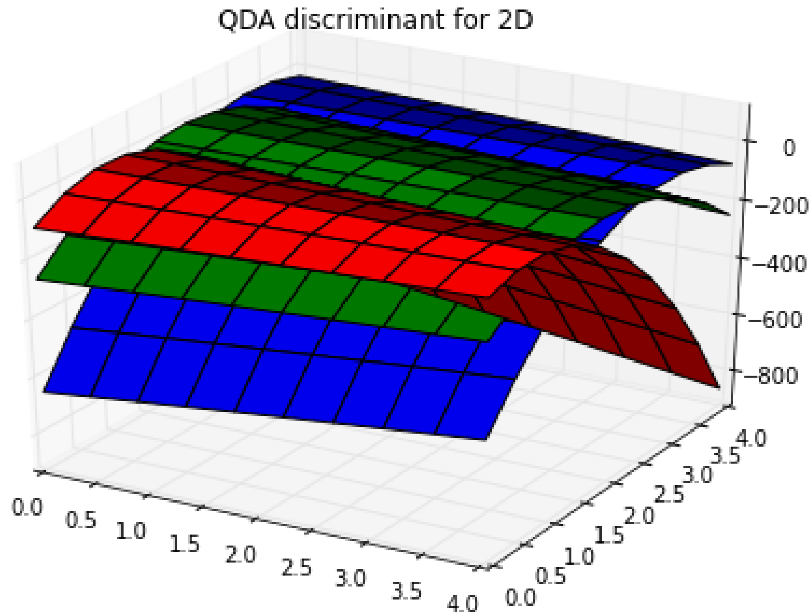

Figure 11: Plot

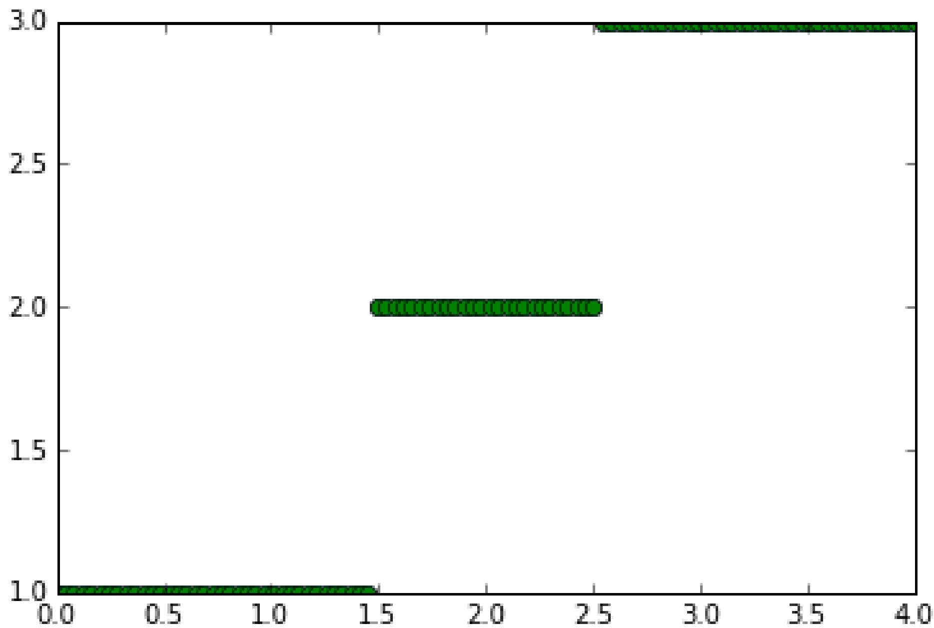

Figure 12: Plot