HW1s

October 6, 2016

```
In [1]: import numpy as np
        np.random.seed(0)
        import mltools as ml
        import matplotlib.pyplot as plt   # use matplotlib for plotting with inline plots
        %matplotlib inline
```

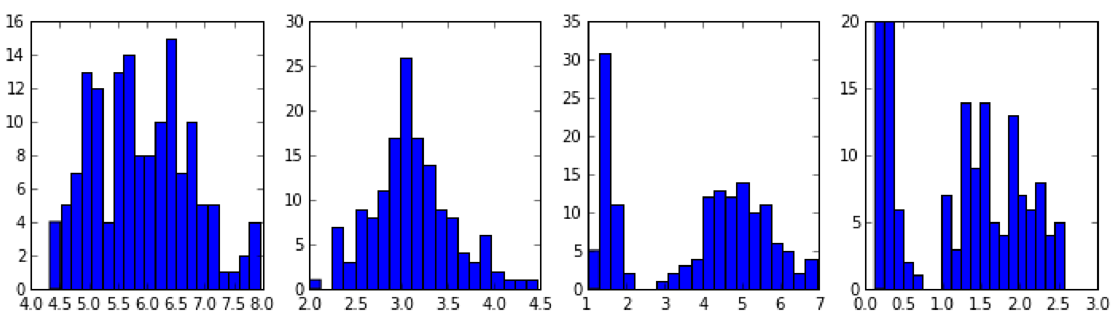
1 P1: Data Exploration

```
In [2]: iris = np.genfromtxt("data/iris.txt",delimiter=None)
        print iris.shape
```

```
(148, 5)
```

```
In [3]: Y = iris[:,-1]
        X = iris[:,0:-1]
        m,n = X.shape
```

```
In [4]: plt.rcParams['figure.figsize'] = (12.0, 3.0)
        fig,ax = plt.subplots(1,4)
        for i in range(n):
            ax[i].hist( X[:,i], bins=20)
        plt.rcParams['figure.figsize'] = (6.0, 4.0)
```



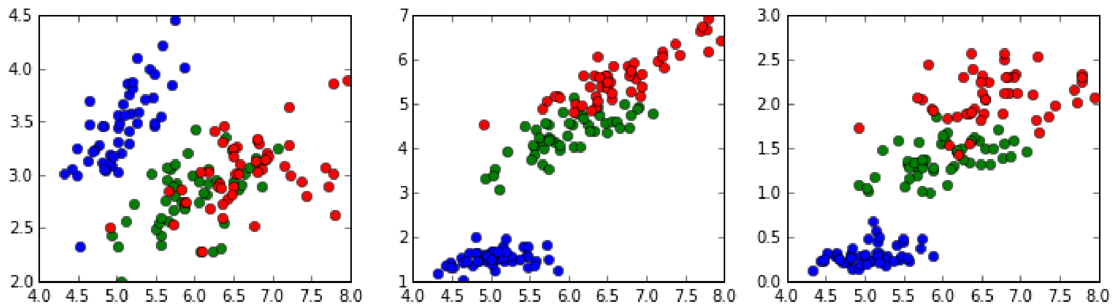

```
In [5]: print 'Mean:', np.mean(X,axis=0)
        print 'Var: ', np.var(X,axis=0)
        print 'Std: ', np.std(X,axis=0)
```

```
Mean: [ 5.90010376  3.09893092  3.81955484  1.25255548]
Var:  [ 0.694559    0.19035057  3.07671634  0.57573564]
Std:  [ 0.83340207  0.43629184  1.75405711  0.75877246]
```

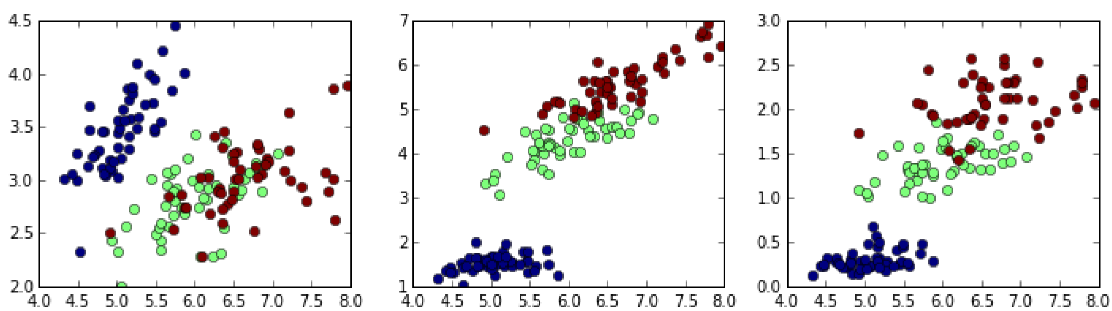
```
In [6]: Xn = X - np.mean(X,axis=0)
        print 'Normed Mean:', np.mean(Xn,axis=0)
        # The mean is now zero (up to numerical precision)
```

```
Normed Mean: [  2.16043399e-15   3.75075346e-16  -6.42128993e-16   3.75075346e-16]
```

```
In [7]: plt.rcParams['figure.figsize'] = (12.0, 3.0)
        fig,ax = plt.subplots(1,3)
        colors = ['b','g','r']
        for i in range(1,n):
            for c in np.unique(Y):
                ax[i-1].plot( X[Y==c,0], X[Y==c,i], 'o', color=colors[int(c)] )
```



```
In [8]: # Alternative, "simpler" version using provided tools
        plt.rcParams['figure.figsize'] = (12.0, 3.0)
        fig,ax = plt.subplots(1,3)
        for i in range(1,n):
            ml.plotClassify2D(None, X[:,[0,i]],Y,axis=ax[i-1])
```

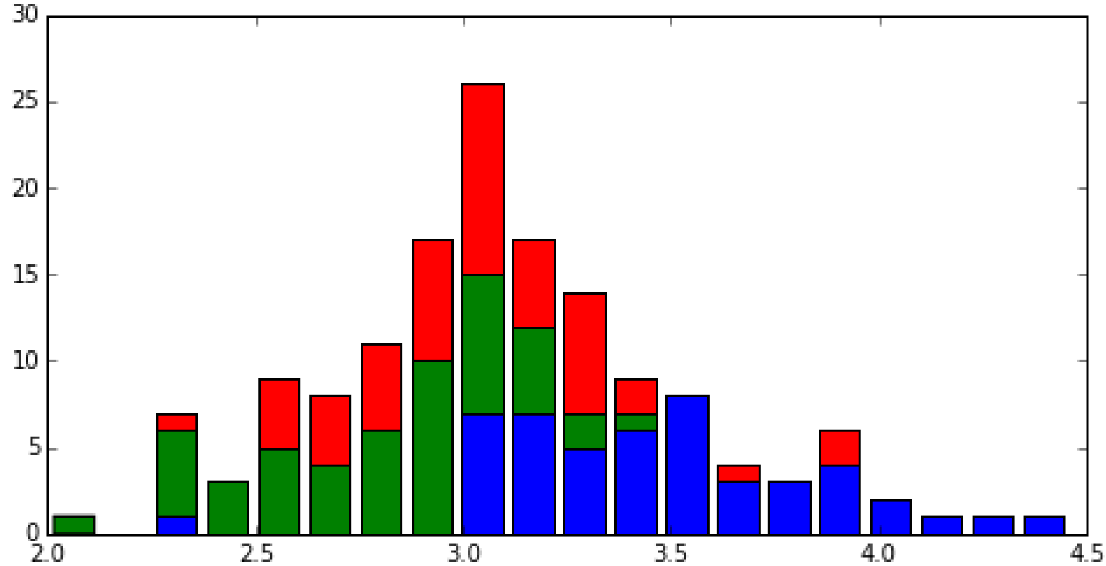


```
In [9]: # You can also produce similar plots using
        # paths = plt.scatter(X[:,1],X[:,2],c=Y)    # scatter function
        # but I personally don't like how they look, so I do it this way.
```

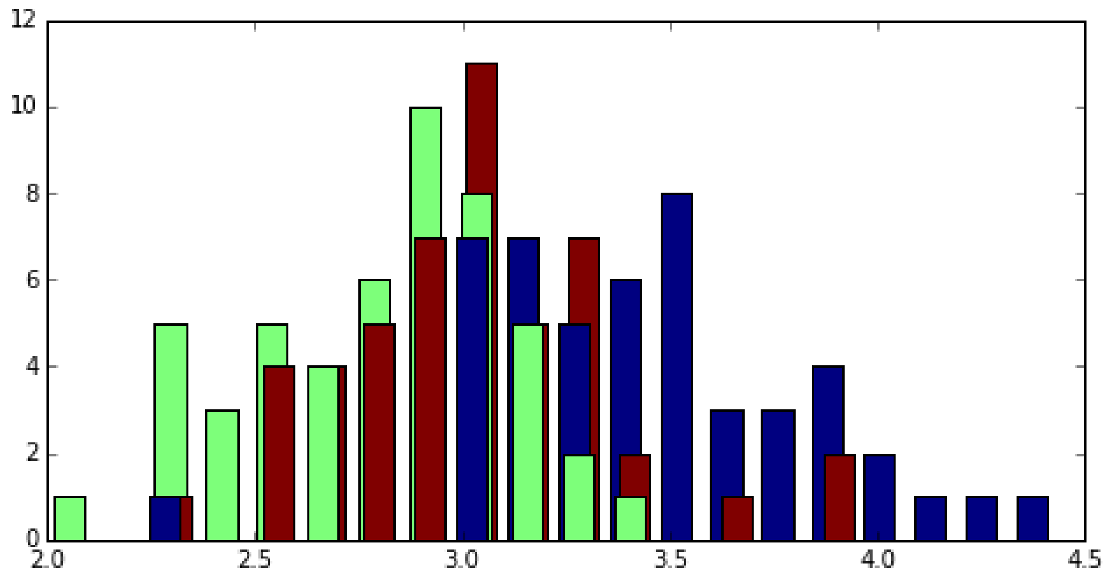
1.0.1 Not required

I'll also illustrate the class-specific histograms, since you may find these useful later on. We can do two different styles; a histogram of the data together, with class-specific fill colors ("plt.hist"), or a collection of class-specific histograms on the same plot ("ml.histy").

```
In [10]: plt.rcParams['figure.figsize'] = (8.0, 4.0)
         n,bins,patches = plt.hist( [X[Y==c,1] for c in np.unique(Y)] , bins=20, histtype='barstacked')
```




```
In [11]: plt.rcParams['figure.figsize'] = (8.0, 4.0)
         ml.histy(X[:,1],Y, bins=20)
```



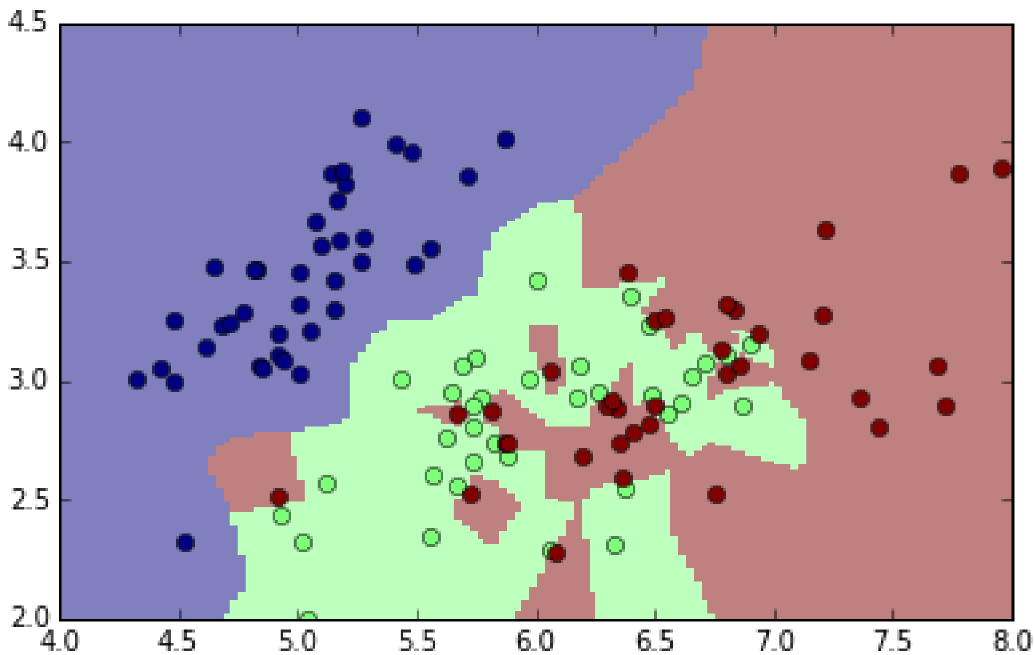
2 P2: kNN Predictions

Start by loading the data again, etc.

```
In [12]: iris = np.genfromtxt("data/iris.txt",delimiter=None)
         X,Y = iris[:,0:4], iris[:,4]
         X,Y = ml.shuffleData(X,Y)
         Xtr, Xva, Ytr, Yva = ml.splitData(X,Y, .75)
```

Now, let's plot the k-nearest neighbor decision boundary using only the first two features:

```
In [13]: knn = ml.knn.knnClassify()
         knn.train(Xtr[:,0:2],Ytr)
         knn.K = 1
         ml.plotClassify2D(knn, Xtr[:,0:2],Ytr)
```

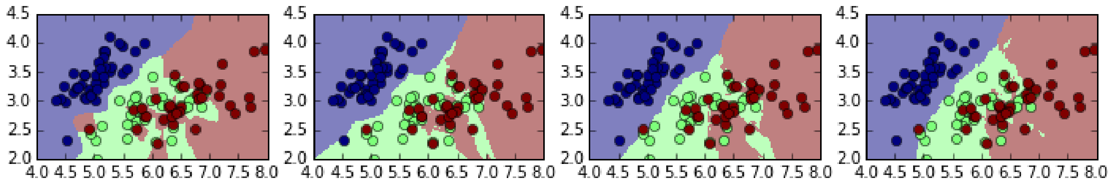


Let's compute error rates at various values of K, and visualize the decision functions as well:

```
In [14]: plt.rcParams['figure.figsize'] = (12.0, 5.0)
         fig,ax = plt.subplots(1,4)

         for i,k in enumerate([1, 5, 10, 20]):
             knn = ml.knn.knnClassify()  #!!! TODO: name
             knn.train(Xtr[:,0:2],Ytr)
             knn.K = k
             print "K=",knn.K, "\t Err (Train):",knn.err(Xtr[:,0:2],Ytr), "\t Err (Val):", knn.err(Xva[
             ml.plotClassify2D(knn, Xtr[:,0:2],Ytr, axis=ax[i])

K= 1           Err (Train): 0.0          Err (Val): 0.297297297297
K= 5           Err (Train): 0.135135135135          Err (Val): 0.27027027027
K= 10           Err (Train): 0.18018018018          Err (Val): 0.378378378378
K= 20           Err (Train): 0.279279279279          Err (Val): 0.189189189189
```



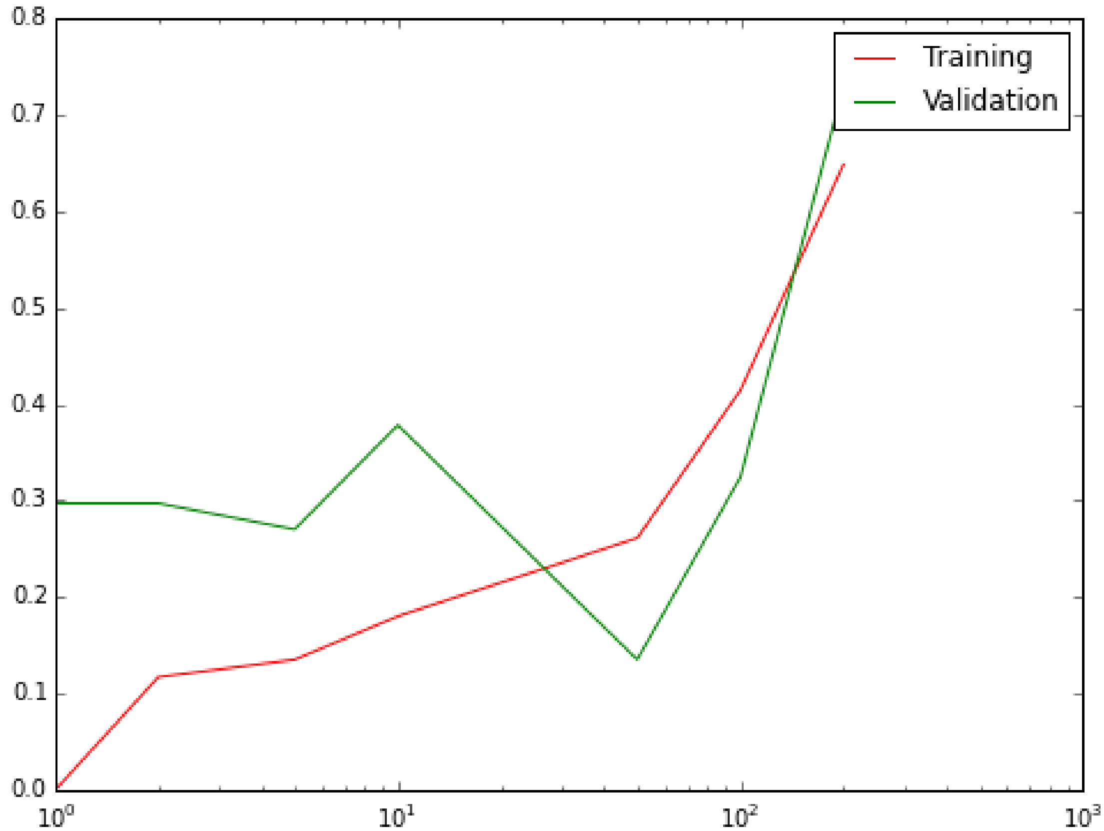
Now, compute the error rate at more values of K, and plot both the training error rate and validation error rates:

```
In [15]: #plt.figure( figsize=(8.0,6.0), )
         plt.rcParams['figure.figsize'] = (8.0, 6.0)
         fig,ax = plt.subplots(1,1)

         knn = ml.knn.knnClassify(Xtr[:,0:2],Ytr)
         k_values = [1, 2, 5, 10, 50, 100, 200]
         errTr = np.zeros((len(k_values),))
         errVa = errTr.copy()
         for i,k in enumerate(k_values):
             knn.K = k
             errTr[i] = knn.err(Xtr[:,0:2],Ytr)
             errVa[i] = knn.err(Xva[:,0:2],Yva)

         ax.semilogx(k_values,errTr,'r-',k_values,errVa,'g-')
         ax.legend(['Training','Validation'])
         print "Training and validation error as a function of K:"
```

```
Training and validation error as a function of K:
```



Based on this plot, k = 50 had the lowest validation error, so I would most likely choose that. You can also see evidence of overfitting (k = 1..10; low training error but high validation error) and of underfitting (k = 100 or more; similar, high training and validation errors).

Your plots may be a bit different, and end up with slightly different values of k, but most likely the trend is similar – at low k, training error is much lower than validation error, suggesting overfitting; at high k, they are similar but high, suggesting underfitting.

3 P3: Bayes Classifiers

You can most easily do this problem by hand, but I'll put it in the Python notebook.

```
In [16]: #(a)
         p_y = 4.0/10;    # p(y) = 4/10
         # p(xi | y=-1)
         p_x1_y0 = 3.0/6;
         p_x2_y0 = 5.0/6;
         p_x3_y0 = 4.0/6;
         p_x4_y0 = 5.0/6;
         p_x5_y0 = 2.0/6;
         # p(xi | y=+1)
         p_x1_y1 = 3.0/4;
         p_x2_y1 = 0.0/4;
         p_x3_y1 = 3.0/4;
```

```
p_x4_y1 = 2.0/4;
p_x5_y1 = 1.0/4;
```

In [17]:
```
# (b)
f_y1_00000 = p_y*(1-p_x1_y1)*(1-p_x2_y1)*(1-p_x3_y1)*(1-p_x4_y1)*(1-p_x5_y1)
print "f_y1_00000 = ",f_y1_00000

f_y0_00000 = (1-p_y)*(1-p_x1_y0)*(1-p_x2_y0)*(1-p_x3_y0)*(1-p_x4_y0)*(1-p_x5_y0)
print "f_y0_00000 = ",f_y0_00000

if (f_y1_00000 > f_y0_00000):
    print "Predict class +1"
else:
    print "Predict class -1"

print "\n\n"

f_y1_11010 = p_y*(p_x1_y1)*(p_x2_y1)*(1-p_x3_y1)*(p_x4_y1)*(1-p_x5_y1)
print "f_y1_11010 = ",f_y1_11010

f_y0_11010 = (1-p_y)*(p_x1_y0)*(p_x2_y0)*(1-p_x3_y0)*(p_x4_y0)*(1-p_x5_y0)
print "f_y0_11010 = ",f_y0_11010

if (f_y1_11010 > f_y0_11010):
    print "Predict class +1"
else:
    print "Predict class -1"

print "\n"
```

```
f_y1_00000 =  0.009375
f_y0_00000 =  0.00185185185185
Predict class +1



f_y1_11010 =  0.0
f_y0_11010 =  0.0462962962963
Predict class -1
```

In [18]:
```
# (c)
# p(y1|11010) =
print "p(y=1|11010) = ", f_y1_11010 / (f_y1_11010 + f_y0_11010)

print "\n"
# For the other pattern (not required), p(y1|00000) =
print "p(y=1|00000) =", f_y1_00000 / (f_y1_00000 + f_y0_00000)
```

```
p(y=1|11010) =  0.0


p(y=1|00000) = 0.835051546392
```

(d) A Bayes classifier using a joint distribution model for $p(x|y = c)$ would have $2^5 - 1 = 31$ degrees of freedom (independent probabilities) to estimate; here we have only 6 and 4 data points respectively. So such a model would be extremely unlikely to generalize well to new data.