Approximate Inference and Clustering

April 12, 2017

1 1. Approximate Inference and Logistic Regression

Implement regression solvers that use Newton-Rhapson to find the MLE and MAP estimates of the weights, respectively.

1.0.1 (i) Implement a regression solver using maximum likelihood.

In [16]:
```
import numpy
import math

def MLElogReg(features, targets, epsilon = math.exp(-20), prior = None):
    """ Estimates MLE logistic regression weights using Newton-Rhapson. """

    # generate an initial weight vector of random values between 0 and 1
    weights = numpy.zeros(features.shape[1])
    # initialize change to none
    delta = None
    priorprob = 1

    # update the weights until delta is less than epsilon
    while not delta or (delta >= epsilon):
        # if there is a prior variance given, update the prior probability of the weigh
        if prior is not None:
            priorprob = (math.sqrt(2*math.pi*numpy.linalg.det(prior))
                         *numpy.exp(-numpy.dot(numpy.transpose(weights), numpy.dot(prio
        # find the gradient vector according to the equation we discussed in class
        gradient = priorprob*(numpy.dot(numpy.transpose(targets),features) -
                   numpy.dot(numpy.transpose(features), 1/(1+numpy.exp(-numpy.dot(feat
        # use that to find the probability of each instance
        P = 1/(1+numpy.exp(-numpy.dot(features, weights)))
        # which can be used to calculate the Hessian matrix
        Hessian = -priorprob*numpy.dot(numpy.transpose(features), numpy.dot(numpy.dot(P
        # calculate the change in the weight vector
        d = numpy.dot(numpy.linalg.inv(Hessian), gradient)
        # calculate the total change to compare to epsilon
        delta = numpy.sum(numpy.absolute(d))
        # generate the new weight vector
```

```
            weights = weights - d

        # return the final optimized weights
        return weights
```

1.0.2 (ii) Implement a regression solver using a Normal prior. Approximate the posterior using Laplace approximation.

```
In [8]: def BayeslogReg(features, targets, priorvar, epsilon = math.exp(-20)):
            """ Estimate logistic regression weights from a Normal prior. """

            # first we'll use Newton-Rhapson, as above to find the MAP estimates of w, which ser
            MAPw = MLElogReg(features, targets, epsilon, prior = priorvar)

            # the posterior variance is the negative inverse Hessian
            # to find that first we must find the probability of each instance to calculate the
            P = 1/(1+numpy.exp(-numpy.dot(features, MAPw)))
            # once I find that, I can calcualte the Hessian as discussed in class
            Hessian = -numpy.linalg.inv(priorvar) - numpy.dot(numpy.transpose(features), numpy.d

            # and now I can return the estimated posterior mean and variance
            return MAPw, -numpy.linalg.inv(Hessian)
```

1.1 a) Test your solver on some of the binary classification datasets we have used before.

First, I need to be able to predict the probability that an instance belongs to class 1.

```
In [3]: def predictMLE(features, weights):
            """ Calculate the probability that each instance is in class 1 using the provdied we
            return 1/(1+numpy.exp(-numpy.dot(features, numpy.transpose(weights))))


        def predictBayes(features, mu, sigma, trials = 100):
            """ Sample weights from the approximated posterior, compute probabilities, and avera

            # numpy array to hold the probability that an instance is in class 1 given each gene
            prob1 = numpy.empty((features.shape[0], trials))

            # generate the user provided number of weights from the distribution and fit the dat
            for t in range(trials):
                # generate a new set of weights from the distribution
                weights = numpy.random.multivariate_normal(mu, sigma)
                # calculate the probability that each instance is a 1 using the same equation as
                prob1[:, t] = predictMLE(features, weights)

            # average over all of the trials for each instance
            return numpy.mean(prob1, axis = 1)
```

Now I can classify instances using MLE and a prior.

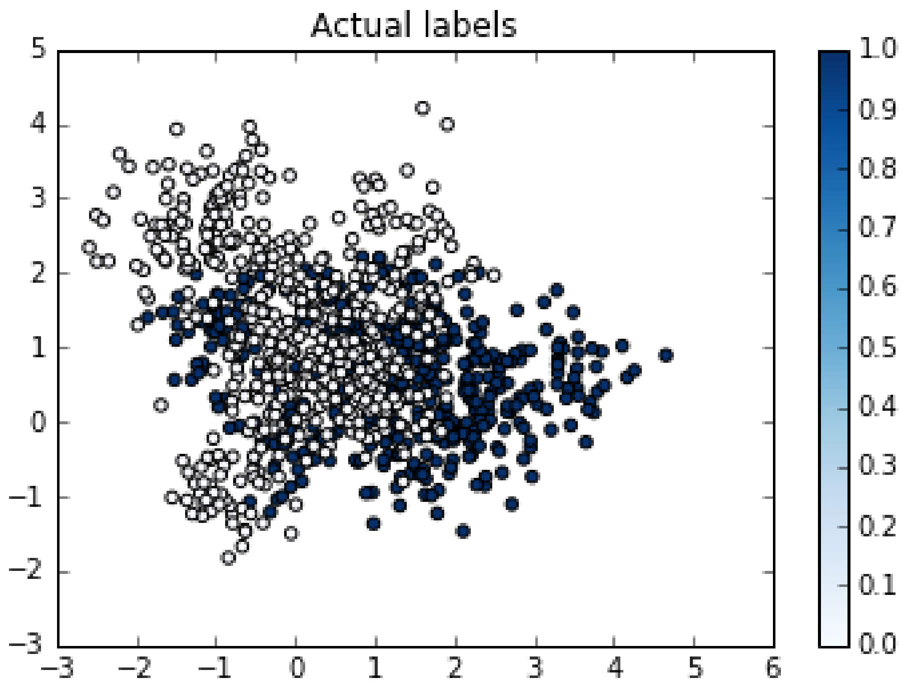
```
In [19]: # load up some binary classification data sets and test on those
         for dataset in ["S1", "S2", "cancer"]:
             print(dataset)
             # clean up the cancer data set for use
             if dataset == "cancer":
                 testset = numpy.loadtxt("cancer_test.csv", delimiter = ",", skiprows = 1)[:, 1:
                 trainset = numpy.loadtxt("cancer_train.csv", delimiter = ",", skiprows = 1)[:,
             else:
                 testset = numpy.loadtxt(dataset+"test.csv", delimiter = ",")
                 trainset = numpy.loadtxt(dataset+"train.csv", delimiter = ",")
             # make sure the labels are 1s and 0s
             testset[:, 0] = 1*(testset[:, 0] == 1)
             trainset[:, 0] = 1*(trainset[:, 0] == 1)
             # calculate the MLE weights
             MLEweights = MLElogReg(trainset[:, 1:], trainset[:, 0])
             # find the probability each instance is in class 1 with MLE
             MLEclasses = predictMLE(testset[:, 1:], MLEweights)
             # print out the misclassification rate
             print("MLE misclassification: " + str(numpy.mean(1*(1*(MLEclasses > .5) != testset[
             # find the Normal parameters
             Bmu, Bsigma = BayeslogReg(trainset[:, 1:], trainset[:, 0],
                                       numpy.amax(MLEweights)*numpy.identity(trainset.shape[1]-1
             # approximate the posterior predictive probability of each test instance
             Bclasses = predictBayes(testset[:, 1:], Bmu, Bsigma)
             # print out the misclassification rate
             print("Bayesian misclassification: " + str(numpy.mean(1*(1*(Bclasses > .5) != tests
             print()
```

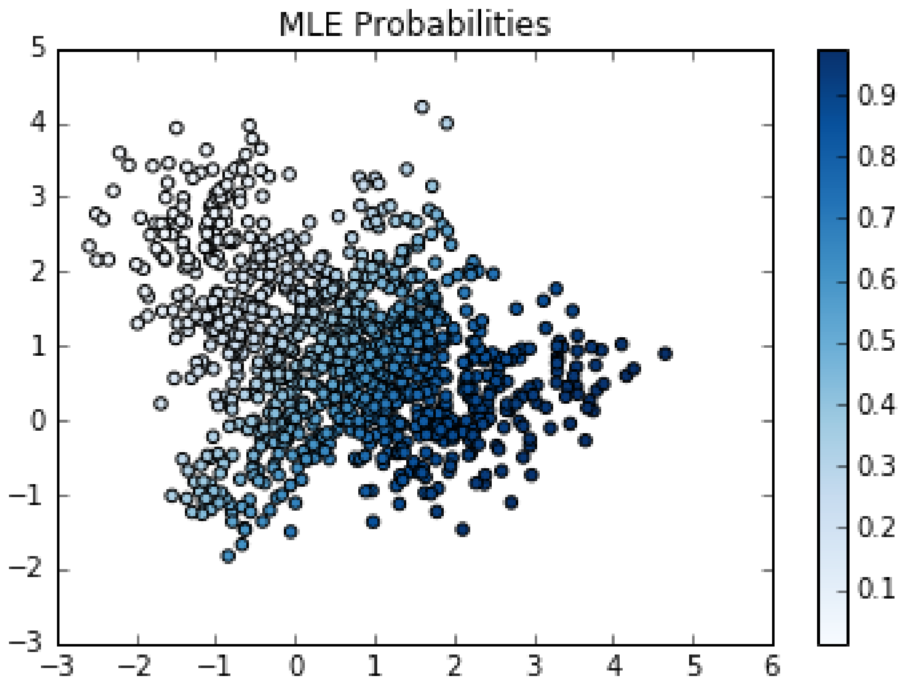
```
S1
MLE misclassification: 0.225
Bayesian misclassification: 0.224

S2
MLE misclassification: 0.283
Bayesian misclassification: 0.283

cancer
MLE misclassification: 0.145922746781
Bayesian misclassification: 0.141630901288
```

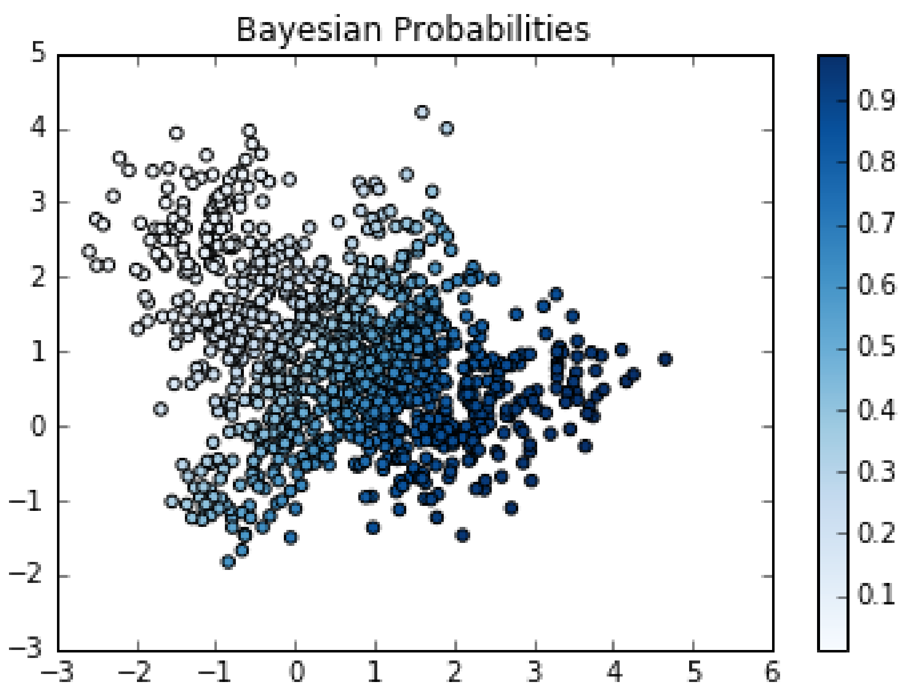
1.2 b) For the 2D synthetic data, produce a plot of the data, color-coded by class and by predicted class probabilities according to the MLE and Bayesian solvers.

```
In [20]: import matplotlib.pyplot as plt
```

```python
def plotProbs(features, classes, title):
    """ Plots instances by their features, colored according to the probability of belo

    plt.scatter(features[:, 0], features[:, 1], c = classes, cmap = "Blues")
    plt.colorbar()
    plt.title(title)
    plt.show()


# load the data and display the testset
S2test = numpy.loadtxt("S2test.csv", delimiter = ",")
S2train = numpy.loadtxt("S2train.csv", delimiter = ",")
S2test[:, 0] = numpy.equal(S2test[:, 0], 1)*1
S2train[:, 0] = numpy.equal(S2train[:, 0], 1)*1
plotProbs(S2test[:, 1:], S2test[:, 0], "Actual labels")

# find the MLE weights and display the probabilities based on those
weights = MLElogReg(S2train[:, 1:], S2train[:, 0])
MLEprobs = predictMLE(S2test[:, 1:], weights)
plotProbs(S2test[:, 1:], MLEprobs, "MLE Probabilities")

# find the Bayesian probabilities
mu, sigma = BayeslogReg(S2train[:, 1:], S2train[:, 0], numpy.amax(weights)*numpy.identi
Bprobs = predictBayes(S2test[:, 1:], mu, sigma)
plotProbs(S2test[:, 1:], Bprobs, "Bayesian Probabilities")
```




MLE Probabilities


Bayesian Probabilities