In [1306]:
```
import numpy as np
import matplotlib.pyplot as plt

#sigmoid and its derivative
def sigmoid(t):
    return 1 / (1 + np.exp(-t))

def sigmoid_derivative(p):
    return p * (1 - p)

np.random.seed(100)


class NeuralNetwork:
    def __init__(self,x=[[]],y=[],numLayers=2,numNodes=2,eta=0.1,maxIter=10000):


        self.lr = eta
        self.numLayers = numLayers
        self.numNodes = numNodes

        self.weights = [np.random.uniform(-1,1,(x.shape[1],numNodes))] #create the weights from the inp

        for i in range(1,numLayers-1):
            self.weights.append(np.random.uniform(-1,1,(numNodes,numNodes))) #create the random weights

        self.weights.append(np.random.uniform(-1,1,(numNodes,1))) #create weights from final layer to o

        self.outputs = np.zeros(y.shape)    #creates an output array of similar size to y

        self.bias = [np.ones(numNodes)] #create bias nodes to the first layer, all 1's
        for i in range(numLayers-1):    #create bias in btwn hidden layers
            self.bias.append(np.ones(numNodes))
        self.bias.append(np.ones(y.shape))#create bias node to last layer, to output, all 1's

        self.train(learningRate=eta,maxIterations=maxIter)

    def train(self,learningRate,maxIterations):
        for i in range(1,maxIterations):
            self.feedforward()
            self.backprop()
```

```python
    def predict(self,x=[]):
        test.feedforward()
        return self.outputs


    def feedforward(self):
        self.layerdata = []

        #this adds in the input layer to the first hidden layer
        self.to_first_layer = sigmoid(np.dot(x, self.weights[0]) + self.bias[0])
        self.layerdata.append(self.to_first_layer) #

        #this adds from hidden layer to hidden layer
        for i in range(1,self.numLayers-1):
            self.layerdata.append(sigmoid(np.dot(self.layerdata[i-1], self.weights[i] + self.bias[i])))

        #this adds from the last hidden layer to output
        self.to_output_layer =  sigmoid(np.dot(self.layerdata[-1], self.weights[-1]) + self.bias[-1])
        self.layerdata.append(self.to_output_layer)
        self.outputs = (self.to_output_layer)


    def backprop(self):

        #error from the actual and predicted output
        self.error = (y - self.layerdata[-1])

        #update the weights to output
        self.output_layer_gradient = sigmoid_derivative(self.layerdata[-1])
        self.hidden_delta = self.error * self.output_layer_gradient #output delta
        self.weights[-1] += np.dot(self.to_first_layer.T, self.hidden_delta) * self.lr

        #loop the hidden layers
        j = -1
        for i in range(1, self.numLayers-1,):
            self.hidden_layer_gradient = sigmoid_derivative(self.layerdata[j-1])
            self.hidden_error = np.dot(self.hidden_delta,self.weights[j].T)
            self.hidden_delta = self.hidden_error * self.hidden_layer_gradient
            self.weights[j-1] += np.dot(self.layerdata[j].T, self.hidden_delta) * self.lr
            j = j-1

        #update the input's weights
        self.hidden_layer_gradient = sigmoid_derivative(self.to_first_layer)
        self.hidden_error = np.dot(self.hidden_delta,self.weights[1].T)
        self.hidden_delta = self.hidden_error * self.hidden_layer_gradient
        self.weights[0] += np.dot(x.T, self.hidden_delta) * self.lr
```

In [1310]:

```
x=np.array([[1,0,1,0],[1,0,1,1],[0,1,0,1]])
y=np.array([[1],[1],[0]])

test = NeuralNetwork(x,y, numLayers=4, numNodes=2, maxIter=100000)
print(test.predict(y))
```

```
[[0.99781205]
 [0.99746123]
 [0.00595618]]
```

Question 2

In [1142]:

```
from sklearn.neural_network import MLPClassifier
import numpy as np
from sklearn.model_selection import cross_val_score
import pandas as pd
import time
import datetime
import matplotlib.pyplot as mp
from pylab import show
```

In [721]:

```
data = np.loadtxt("transfusion.csv",delimiter=",")

features = []
donated = []

for row in data:
    features.append(row[:4])
    donated.append(row[-1])
```

```
In [800]: #The following parameters for layers and nodes were modified in accordance with a Piazza post stating to
          #both blood transfusion and digits data
          Neural_Networks = {} #create a dictionary of the neural network results

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(2))
          Neural_Networks['Layers=1, Nodes=2'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(2,2))
          Neural_Networks['Layers=2, Nodes=2'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(2,2,2,2,2))
          Neural_Networks['Layers=5, Nodes=2'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(2,2,2,2,2,2,2,2,2,2))
          Neural_Networks['Layers=10, Nodes=2'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          #-------------------------------------------------------------------------------------------

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(5))
          Neural_Networks['Layers=1, Nodes=5'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(5,5))
          Neural_Networks['Layers=2, Nodes=5'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(5,5,5,5,5))
          Neural_Networks['Layers=5, Nodes=5'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(5,5,5,5,5,5,5,5,5,5))
          Neural_Networks['Layers=10, Nodes=5'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

          #-------------------------------------------------------------------------------------------

          mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                              hidden_layer_sizes=(10))
```

```python
Neural_Networks['Layers=1, Nodes=10'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(10,10))
Neural_Networks['Layers=2, Nodes=10'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(10,10,10,10,10))
Neural_Networks['Layers=5, Nodes=10'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(10,10,10,10,10,10,10,10,10,10))
Neural_Networks['Layers=10, Nodes=10'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

#-------------------------------------------------------------------------------------------

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(50))
Neural_Networks['Layers=1, Nodes=50'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(50,50))
Neural_Networks['Layers=2, Nodes=50'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(50,50,50,50,50))
Neural_Networks['Layers=5, Nodes=50'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(50,50,50,50,50,50,50,50,50,50))
Neural_Networks['Layers=10, Nodes=50'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

#-------------------------------------------------------------------------------------------

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(100))
Neural_Networks['Layers=1, Nodes=100'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                    hidden_layer_sizes=(100,100))
Neural_Networks['Layers=2, Nodes=100'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                   hidden_layer_sizes=(100,100,100,100,100))
Neural_Networks['Layers=5, Nodes=100'] = [cross_val_score(mlp, features, donated, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=2000, alpha=0, solver="adam",
                   hidden_layer_sizes=(100,100,100,100,100,100,100,100,100,100))
Neural_Networks['Layers=10, Nodes=100'] = [cross_val_score(mlp, features, donated, cv=10).mean()]
```

In [851]:
```
print("Parameters\t\tAccuracy")
for key,val in Neural_Networks.items():
    print("{}\t{}".format(key,val))
```

```
Parameters              Accuracy
Layers=1, Nodes=2       [0.5]
Layers=2, Nodes=2       [0.653945945945946]
Layers=5, Nodes=2       [0.762054054054054]
Layers=10, Nodes=2      [0.762054054054054]
Layers=1, Nodes=5       [0.5453333333333333]
Layers=2, Nodes=5       [0.7348108108108108]
Layers=5, Nodes=5       [0.7340540540540541]
Layers=10, Nodes=5      [0.714054054054054]
Layers=1, Nodes=10      [0.654054054054054]
Layers=2, Nodes=10      [0.658054054054054]
Layers=5, Nodes=10      [0.6559999999999999]
Layers=10, Nodes=10     [0.7633873873873874]
Layers=1, Nodes=50      [0.41796396396396396]
Layers=2, Nodes=50      [0.7274414414414414]
Layers=5, Nodes=50      [0.783405405405405 4]
Layers=10, Nodes=50     [0.7620720720720721]
Layers=1, Nodes=100     [0.7500540540540539]
Layers=2, Nodes=100     [0.5714234234234234]
Layers=5, Nodes=100     [0.714108108108108]
Layers=10, Nodes=100    [0.7327747747747747]
Layers=1, Nodes=1       [0.8872311827956988]
```

The neural network with the highest accuracy is the one with 5 layers and 50 nodes, coming in at 78.34%.

In [843]:
```
data = np.loadtxt("data.csv")

#shuffle the data and select training and test data
np.random.seed(100)
np.random.shuffle(data)

features = []
digits = []

for row in data:
    if(row[0]==1 or row[0]==5):
        features.append(row[1:]) #add in remaining values past the first (this is a matrix)
        digits.append(str(row[0])) #add the 1 or the 5 to digits

#select the proportion of data to use for training
numTrain = int(len(features)*.2) #we are training on 20% of the data

trainFeatures = features[:numTrain] #we train on the first 20% of the data
testFeatures = features[numTrain:]  #we test on the remaining 80%
trainDigits = digits[:numTrain]     #we train on the first 20% of the data
testDigits = digits[numTrain:]      #we test on the remaining 80%
```

```
In [844]: X = []
          Y = []
          simpleTrain = []
          colors = []
          for index in range(len(trainFeatures)):
              X.append((sum(trainFeatures[index])/256)**2) #mean intensity squared
              Y.append(sum((trainFeatures[index][:128]- trainFeatures[index][128:])**2)/256) #horiz symmetry
              simpleTrain.append([(sum(trainFeatures[index])/256)**2,sum((trainFeatures[index][:128]- trainFeature
              if(trainDigits[index]=="1.0"): #if the digit is 1, it is blue. else, it is red (only 1's and 5's)
                  colors.append("b")
              else:
                  colors.append("r")
          #normalization of X and Y
          normX = [2*((i - min(X)) / (max(X) - min(X)))-1 for i in X]
          normY = [2*((i - min(Y)) / (max(Y) - min(Y)))-1 for i in Y]

          normTrain = []
          for i in range(len(normX)):
              normTrain.append([normX[i],normY[i]])
```

```
In [883]: HW1NN = {}
          runtimes = []
          # #Max iteration was brought up to 3000 from 2000 because it was stated that
          # #there was no convergence. This increase allowed it to find a convergence.

          nodes = [2,5,10,50,100]
          for i in nodes:
              mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=3000, alpha=0, solver="adam",
                                  hidden_layer_sizes=(i))
              key_name = ('Layers=1, Nodes:'+ str(i))
              start = time.time()
              HW1NN[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
              end = time.time()
              runtimes.append(end-start)
```

In [892]:
```
for i in nodes:
    mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=3000, alpha=0, solver="adam",
                   hidden_layer_sizes=(i,i))
    key_name = ('Layers=2, Nodes:'+ str(i))
    start = time.time()
    HW1NN[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
    end = time.time()
    runtimes.append(end-start)
```

In [893]:
```
for i in nodes:
    mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=3000, alpha=0, solver="adam",
                   hidden_layer_sizes=(i,i,i,i,i))
    key_name = ('Layers=5, Nodes:'+ str(i))
    start = time.time()
    HW1NN[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
    end = time.time()
    runtimes.append(end-start)
```

In [894]:
```
for i in nodes:
    mlp = MLPClassifier(activation="relu", epsilon=.001, max_iter=3000, alpha=0, solver="adam",
                   hidden_layer_sizes=(i,i,i,i,i,i,i,i,i,i))
    key_name = ('Layers=10, Nodes:'+ str(i))
    start = time.time()
    HW1NN[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
    end = time.time()
    runtimes.append(end-start)
```

```
In [895]: index = 0
          print("Parameters\t\t\tAccuracy\t\tRuntimes")
          for key,val in HW1NN.items():
              print("{}\t\t{}".format(key,val),"\t{}".format(runtimes[index]))
              index +=1
```

```
Parameters                      Accuracy                Runtimes
Layers=1, Nodes:2               [0.8869758064516129]    8.013168096542358
Layers=1, Nodes:5               [0.9612836021505377]    5.71295428276062
Layers=1, Nodes:10              [0.9870900537634408]    5.377238035202026
Layers=1, Nodes:50              [0.9902150537634409]    4.6369359493255615
Layers=1, Nodes:100             [0.9902150537634409]    5.18547511100769
Layers=2, Nodes:2               [0.8968077956989247]    8.361129999160767
Layers=2, Nodes:5               [0.9838642473118279]    8.287352085113525
Layers=2, Nodes:10              [0.9934408602150538]    4.783476829528809
Layers=2, Nodes:50              [0.9902150537634409]    4.840544939041138
Layers=2, Nodes:100             [0.9934408602150538]    6.524170875549316
Layers=5, Nodes:2               [0.7356182795698925]    3.6592350006103516
Layers=5, Nodes:5               [0.8634005376344085]    6.459857940673828
Layers=5, Nodes:10              [0.9903158602150539]    3.944851875305176
Layers=5, Nodes:50              [0.9902150537634409]    4.232532978057861
Layers=5, Nodes:100             [0.9903158602150539]    6.285758972167969
Layers=10, Nodes:2              [0.6699932795698925]    6.671728849411011
Layers=10, Nodes:5              [0.7710013440860215]    4.306555271148682
Layers=10, Nodes:10             [0.9580577956989247]    6.438693046569824
Layers=10, Nodes:50             [0.9934408602150538]    8.407313823699951
Layers=10, Nodes:100            [0.9579569892473119]    14.461431980133057
```

Short Answer

a: For the lower number of layers, as we increase the number of nodes, we see a decrease in the overall runtime up until we hit 100 nodes, where we see an increase. For the larger layer numbers such as 5 and 10, we actually see fluctuating runtimes. Examples like this are when we have an increase in layers=5 where when we go from 2 nodes to 5 nodes, we double the runtime, but increasing to 10 nodes halves it back down. We see a similar, but inversed effect with layers=10, where from 2 to 5 we lower the runtime, but from 5 to 10 we bring it back to the same runtime as 2. At all points, a node number of 100 looks to always increase the runtime.

b: The optimum result fom the above looks to be 99.34%. This is apparent at several levels of layers and nodes. The following are where this is present:

Layers=2, nodes=10

Layers=2, nodes=100

Layers=10, Nodes=50

c: For this question, we will choose Layers=2, nodes=10 as our optimal model. In the below, we can see that as we increase the learning rate to effectively 1, we have a dramatic drop in accuracy but a large drop off in the runtime. When we consider a very small epsilon, it does not look like it has adverse effects to the runtime (actually decreases) and gives the optimal accuracy at lower values of epsilon.

The max iteration was increased due to the model not being able to find convergence at varying epsilon values.

In [1312]:
```
epsilons = [.001, .01, .1, 1, .0001, .00001]
Q2C = {}
Q2CRuntimes = []
for i in epsilons:
    mlp = MLPClassifier(activation="relu", epsilon=i, max_iter=10000, alpha=0, solver="adam",
                        hidden_layer_sizes=(10,10))
    key_name = ('Layers=2, Nodes:10, Epsilon:' + str(i))
    start = time.time()
    Q2C[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
    end = time.time()
    Q2CRuntimes.append(end-start)
```

In [908]:
```
index = 0
print("Parameters\t\t\t\t\tAccuracy\t\tRuntimes")
for key,val in Q2C.items():
    print("{}\t\t{}".format(key,val),"\t{}".format(Q2CRuntimes[index]))
    index +=1
```

```
Parameters                                      Accuracy                Runtimes
Layers=2, Nodes:10, Epsilon:0.001               [0.9869892473118279]    4.519749164581299
Layers=2, Nodes:10, Epsilon:0.01                [0.9869892473118279]    6.099369049072266
Layers=2, Nodes:10, Epsilon:0.1         [0.8998924731182795]    9.76955795288086
Layers=2, Nodes:10, Epsilon:1           [0.4906922043010752]    0.9823930263519287
Layers=2, Nodes:10, Epsilon:0.0001              [0.9869892473118279]    4.4847471714019775
Layers=2, Nodes:10, Epsilon:1e-05               [0.9934408602150538]    3.945981979370117
```

d: It does look like the neural network is finding the same optimal solution, but the accuracy is consistently varying with each run. When rerunning the models, we can see that the accuracy will fluctuate. This will cause changes in the expected fit, as occasionally the runs will also find better results with different layers and different nodes per neuron.

In [915]:
```
Q2E = {}
mlp = MLPClassifier(activation="relu", epsilon=i, max_iter=10000, alpha=0, solver="adam", early_stopping
        hidden_layer_sizes=(100,100,100,100,100))
key_name = ('With EStop')
Q2E[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]

mlp = MLPClassifier(activation="relu", epsilon=i, max_iter=10000, alpha=0, solver="adam",
        hidden_layer_sizes=(100,100,100,100,100))
key_name = ('No EStop')
Q2E[key_name] = [cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean()]
```

In [919]:
```
index = 0
print("Parameters\t\tAccuracy")
for key,val in Q2E.items():
    print("{}\t\t{}".format(key,val))
    index +=1
```

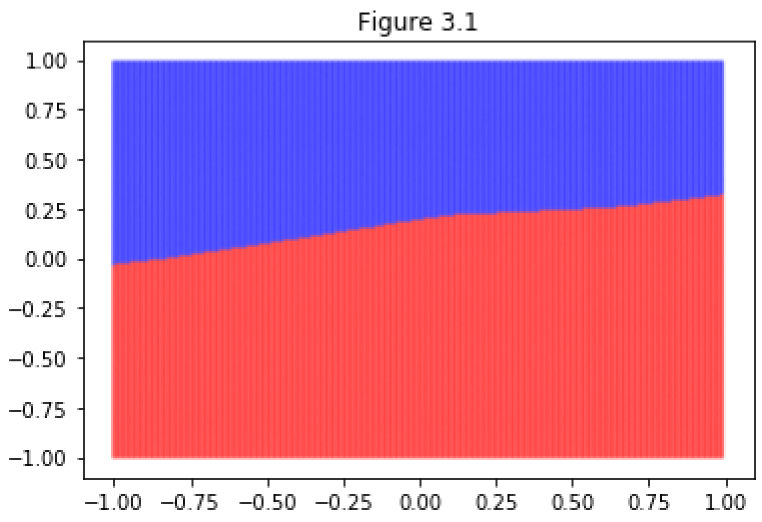
```
Parameters              Accuracy
With EStop              [0.707997311827957]
No EStop                [0.9934408602150538]
```

e: With early stopping, it looks to cause an underfit on the data more than an overfit. From the above, we see that the early stopping hsa an accuracy of 70.7% while without, we have an accuracy of 99.34%. From the information provided in scikit-learn, what early stopping does is it sets a validation set to the side and tests against the validation set. When the validation set is no longer seeing improved results, the iterations of the model will stop. This means that without early stopping, we are actually prone to overfit while we will be closer to a better model with early stopping.

```
In [1146]: model = mlp
model.fit(simpleTrain,trainDigits)

xPred = []
yPred = []
cPred = []
for xP in range(-100,100):
    xP = xP/100.0
    for yP in range(-100,100):
        yP = yP/100.0
        xPred.append(xP)
        yPred.append(yP)
        if(model.predict([[xP,yP]])=="1.0"):
            cPred.append("r")
        else:
            cPred.append("b")

mp.scatter(xPred,yPred,s=3,c=cPred,alpha=.2, )
mp.title("Figure 3.1")
show()
```



Extra Credit

```
In [1324]: import pandas
            from sklearn import model_selection
            from sklearn.ensemble import BaggingClassifier
            from sklearn.tree import DecisionTreeClassifier
```

```
In [1329]: #NN by itself
            mlp = MLPClassifier(activation="relu", epsilon=i, max_iter=10000, alpha=0, solver="adam",
                    hidden_layer_sizes=(100,100,100,100,100))
            print(cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean())

            #Bagged NN
            model = BaggingClassifier(base_estimator=mlp, n_estimators=10)
            results = model_selection.cross_val_score(model, trainFeatures, trainDigits, cv=kfold)
            print(results.mean())

            #simplifying the NN
            #NN by itself
            mlp = MLPClassifier(activation="relu", epsilon=i, max_iter=10000, alpha=0, solver="adam",
                    hidden_layer_sizes=(5,5))
            print(cross_val_score(mlp, simpleTrain, trainDigits, cv=10).mean())

            model = BaggingClassifier(base_estimator=mlp, n_estimators=5)
            results = model_selection.cross_val_score(model, trainFeatures, trainDigits, cv=kfold)
            print(results.mean())
```

```
0.9934408602150538
0.9967741935483871
0.9611827956989247
0.9904249871991808
```

1: The number of bagged neural networks has a positive effect as we increase the total number of bagged neural networks (the n_estimators was increased from 5 to 10 which had a subtle but noticeable effect).

2: The first NN above is more complex while the second is more simplified. The more simplified looks to have less overall accuracy on both fronts of without bagging and with bagging. Because of this, we conclude that more complex models have a better effect.

3: Neural networks are prone to not converge when there are large datasets given a number of iterations and with the layers and neurons per layer. Because of this, sometimes there may be large variance in the results. Bagging helps accomodate for this by averaging out the results and providing a mean accuracy, providing more credibility to the results and not letting it just be based on chance that you come across favorable results.

4: Bagging helps the neural networks achieve a higher percent accuracy, but the cost is a much longer overall runtime (you are running several neural networks and then finding the mean). Because of this, we consider the tradeoff and, if time were not an issue, I would choose this as an optimal neural network.

```
In [ ]:
```