Tyler Bryk

CPE-695 HW4

1.) Find $P(\text{Cancer} \mid \text{Two Positive Tests})$

$P(\text{Cancer}) = 0.008$

$P(\neg\text{Cancer}) = 1 - P(\text{Cancer}) = 1 - 0.008 = 0.992$

$P(+ \mid \text{Cancer}) = 0.98$ $\quad$ $P(- \mid \neg\text{Cancer}) = 0.97$

$P(- \mid \text{Cancer}) = 0.02$ $\quad$ $P(+ \mid \neg\text{Cancer}) = 0.03$

First, we find $P(\text{Cancer} \mid \text{One Positive})$

$$P(\text{Cancer} \mid +) = P(+ \mid \text{Cancer}) \cdot P(\text{Cancer}) \;/\; P(+)$$

Where $P(+) = P(+ \mid \text{Cancer}) \cdot P(\text{Cancer}) + P(+ \mid \neg\text{Cancer}) \cdot P(\neg\text{Cancer})$

$\hookrightarrow = [0.98] \cdot [0.008] + [0.03] \cdot [0.992] = 0.0376$

$P(\text{Cancer} \mid +) = (0.98) \cdot (0.008) \;/\; (0.0376) = \underline{0.2085}$

Now, using $P(\text{Cancer} \mid +) = 0.21$, we find $P(\text{Cancer} \mid ++)$

$$P(\text{Cancer} \mid ++) = P(+ \mid \text{Cancer}) \cdot P(\text{Cancer} \mid +) \;/\; P(++)$$

Where $P(++) = P(+ \mid \text{Cancer})\, P(\text{Cancer} \mid +) + P(+ \mid \neg\text{Cancer})\, P(\neg\text{Cancer} \mid +)$

$\hookrightarrow = [0.98][0.2085] + [0.03][1 - 0.2085] = 0.2281$

$P(\text{Cancer} \mid ++) = (0.98) \cdot (0.2085) \;/\; (0.2281) = 0.8960$

$P(\text{Cancer} \mid ++) = 0.90 = 90\%$

$P(\neg\text{Cancer} \mid ++) = 0.10 = 10\%$

2.) Find 95% Confidence Interval for True Error Rate for $Error_D(h)$ When 85 / 100 are correctly classified.

For 95% Confidence → $\alpha = 0.05$ and $\alpha/2 = 0.025$

Using Probability Table, $Z = 1.960$ for $\alpha/2 = 0.025$

$$CI\ Error_D(h) = \pm 1.96 * \sqrt{Error_D(h)\left(1 - Error_D(h)\right) / n}$$

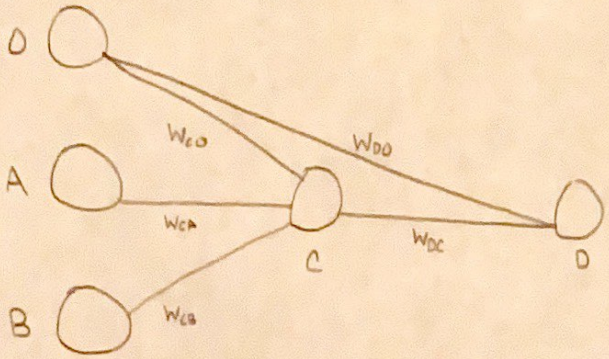
$$\hookrightarrow \left(1 - \frac{85}{100}\right) \pm 1.96 \sqrt{\left(1 - \frac{85}{100}\right)\left(1 - \left(1 - \frac{85}{100}\right)\right) / 100}$$

$$\hookrightarrow 0.15 \pm 1.96 \sqrt{(0.15)(0.85) / 100}$$

$$\hookrightarrow 0.15 \pm 0.06999 \rightarrow 0.15 \pm 0.07$$

95% CI for $Error_D(h)$ is $[0.08, 0.22]$

3.) Use Back Propagation to update Weights after two training iterations.



Sigmod:

$$\sigma(y) = \frac{1}{1+e^{-y}}$$

Training Example 1: A=1 / B=0 / D=1

$O_C = \sigma(0.1(1) + 0.1(0) + 0.1(1)) = \sigma(0.2) = 0.54980$

$O_D = \sigma(0.1(0.5498) + 0.1(1)) = \sigma(0.15498) = 0.53867$ ← Outputs

$\varepsilon_D = 0.53867[(1-0.53867)(1-0.53867)] = 0.11460$

$\varepsilon_C = 0.54980[(1-0.54980)(0.1(0.1146))] = 0.002836$ ← Error

$\Delta W_{DO} = 0.3(0.1146)(1) = 0.0342 \rightarrow W_{DO} = 0.1 + 0.0342 = 0.1342$

$\Delta W_{DC} = 0.3(0.1146)(0.5498) = 0.0189 \rightarrow W_{DC} = 0.1 + 0.0189 = 0.1189$

$\Delta W_{CO} = 0.3(0.002836)(1) = 0.000849 \rightarrow W_{CO} = 0.1 + 0.000849 = 0.100849$

$\Delta W_{CA} = 0.3(0.002836)(1) = 0.000849 \rightarrow W_{CA} = 0.1 + 0.000849 = 0.100849$

$\Delta W_{CB} = 0.3(0.002836)(0) = 0 \rightarrow W_{CB} = 0.1 + 0 = 0.1$

Training Weights after 1 Iteration

3.) Continued

Training Example 2: A = 0 B = 1 / D = 0

$O_C = \sigma(0.100849(0) + 0.1(1) + 0.100849(1)) = \sigma(0.200849) = 0.5500$

$O_D = \sigma(0.1189(0.55) + 0.1342(1)) = \sigma(0.1996) = 0.5497$

$\varepsilon_D = 0.5497\left[(1-0.5497)\cdot(0-0.5497)\right] = -0.1361$

$\varepsilon_C = 0.5500\left[(1-0.5500)\cdot(0.1189)\cdot(-0.1361)\right] = -0.004$

$\Delta W_{DO} = 0.3(-0.1361)(1) + 0.9(0.0342) = -0.01$

$\Delta W_{DC} = 0.3(-0.1361)(0.55) + 0.9(0.0189) = -0.0055$

$\Delta W_{CO} = 0.3(-0.004)(1) + 0.9(0.000849) = -0.0004$

$\Delta W_{CA} = 0.3(-0.004)(0) + 0.9(0.000849) = 0.00036$

$\Delta W_{CB} = 0.3(-0.004)(1) + 0.9(0) = -0.0012$

Training Weights after Second Iteration:

$W_{DO} = 0.1342 + -0.01 = 0.1242$

$W_{DC} = 0.1189 + -0.0055 = 0.1134$

$W_{CO} = 0.100849 + -0.0004 = 0.100449$

$W_{CA} = 0.100849 + 0.00036 = 0.1016$

$W_{CB} = 0.1 + -0.0012 = 0.0988$

TylerBryk_CPE695_HW4

November 5, 2020

1 CPE695 HW4

By: Tyler Bryk

In this assignment, we will explore the titanic dataset and create a neural network with backpropagation to predict the survival rate of passengers on board.

Step 1: Read in Titanic.csv and observe a few samples, some features are categorical and others are numerical. Take a random 80% samples for training and the rest 20% for test.

In this step, we load our Titanic.csv dataset into a Pandas DataFrame. First, we convert values in the sex column to be numerical, female->0 and male->1, then we do the same for the pclass column, 1st->1, 2nd->2, 3rd->3. The age column contains several missing values, so we will impute those missing instances with the median value for the age column. Then, we will split our dataset into 80% training data, and 20% testing data. Lastly, we use a normalizer to scale the data so that it is centered around the mean.

[1]:
```
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, plot_confusion_matrix,␣
↪accuracy_score
```

[2]:
```
# Load Data and Split 80:20
data = pd.read_csv('Titanic.csv', index_col=0)
data['sex'].replace(['female','male'],[0,1], inplace=True)
data['pclass'].replace(['1st','2nd','3rd'],[1,2,3], inplace=True)
data['age'] = data['age'].fillna(data['age'].median())
x = data[['pclass', 'sex', 'age', 'sibsp']].to_numpy()
y = data['survived'].to_numpy()
xTrain, xTest, yTrain, yTest = train_test_split(x, y, test_size=0.2,␣
↪random_state=10413641)
```

[3]:
```
# Normalize Data by Removing Mean and Scaling by Variance
scaler = StandardScaler()
scaler.fit(xTrain)
xTrain = scaler.transform(xTrain)
xTest = scaler.transform(xTest)
```

[4]:
```
# Create a Transposed Version of Data for Custom NN
xTrainT = np.expand_dims(xTrain,  axis=1)
yTrainT = np.expand_dims(yTrain,  axis=1)
yTrainT = np.expand_dims(yTrainT, axis=1)
xTestT  = np.expand_dims(xTest,  axis=1)
yTestT  = np.expand_dims(yTest,  axis=1)
yTestT  = np.expand_dims(yTestT, axis=1)
```

Step 2: Fit a neural network using independent variables 'pclass + sex + age + sibsp' and dependent variable 'survived'. Omit all NA examples. Use 2 hidden layers and set the activation functions for both the hidden and output layer to be the sigmoid function. Set "solver" parameter as either SGD (stochastic gradient descend) or Adam (similar to SGD but optimized performance with mini batches). You can adjust parameter "alpha" for regularization (to control overfitting) and other parameters such as "learning rate" and "momentum" as needed.

2 Design Neural Network from Scratch

In this section, we will attempt to design a neural network entirely from scratch without the use of any libraries. First, we define constructors for a 'Layer' object, then for two types of layers: Fully-Connected, and Activation. The basis for a layer is that it has inputs and outputs, and can call both forward abd backward propagation. Next, we define a network class which is going to be the framework for our network. This class has methods such as 'add' and 'use' which allow more layers to be added to the network, and for the user to specify the activation/loss function for each added layer. Then, the network of course has methods to fit training data and predict on new data. In the last code module, the activation and loss functions are defined. For this network, we will be sticking to a logistic sigmoid activation, and MSE for loss.

[5]:
```
# Define Class Constructors for Layers
class Layer:
    def __init__(self):
        self.input = None
        self.output = None
    def forwardPropagation(self, input):
        raise NotImplementedError
    def backwardPropagation(self, outputError, learningRate):
        raise NotImplementedError

class FCLayer(Layer):
    def __init__(self, inputSize, outputSize):
        self.weights = np.random.rand(inputSize, outputSize) - 0.5
        self.bias = np.random.rand(1, outputSize) - 0.5
```

```
    def forwardPropagation(self, inputData):
        self.input = inputData
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output
    def backwardPropagation(self, outputError, learningRate):
        inputError = np.dot(outputError, self.weights.T)
        weightsError = np.dot(self.input.T, outputError)
        self.weights = self.weights - (learningRate * weightsError)
        self.bias = self.bias - (learningRate * outputError)
        return inputError

class ActivationLayer(Layer):
    def __init__(self, activation, activationPrime):
        self.activation = activation
        self.activationPrime = activationPrime
    def forwardPropagation(self, inputData):
        self.input = inputData
        self.output = self.activation(self.input)
        return self.output
    def backwardPropagation(self, outputError, learningRate):
        return self.activationPrime(self.input) * outputError
```

[6]:
```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.lossPrime = None
    def add(self, layer):
        self.layers.append(layer)
    def use(self, loss, lossPrime):
        self.loss = loss
        self.lossPrime = lossPrime
    def predict(self, inputData):
        result = []
        for i in range(len(inputData)):
            output = inputData[i]
            for layer in self.layers:
                output = layer.forwardPropagation(output)
            result.append(output)
        return result
    def fit(self, xTrain, yTrain, epochs, learningRate):
        for i in range(epochs):
            err = 0
            for j in range(len(xTrain)):
                output = xTrain[j]
                for layer in self.layers:
                    output = layer.forwardPropagation(output)
```

```
                err += self.loss(yTrain[j], output)
                error = self.lossPrime(yTrain[j], output)
                for layer in reversed(self.layers):
                    error = layer.backwardPropagation(error, learningRate)
            err /= len(xTrain)
            if i % 1000 == 0:
              print('Epoch: %d/%d     Error: %f' % (i, epochs, err))
```

[7]:
```
# Error & Activation Functions
def mse(yTrue, yPred):
    return np.mean(np.power(yTrue - yPred, 2))
def msePrime(yTrue, yPred):
    return 2 * (yPred - yTrue) / yTrue.size
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoidPrime(x):
    return np.exp(-x) / (1 + np.exp(-x))**2
```

[8]:
```
# Define Helper Functions for Calculating Accuracy
def hardAssignment(inputData):
    result = []
    for i in inputData:
      if i[0][0] < 0.5:
        assign = 0
      elif i[0][0] >= 0.5:
        assign = 1
      result.append(assign)
    return result
def accuracy(Predictions, GrandTruth):
    samples = len(GrandTruth)
    numCorrect = 0
    for i in range(samples):
      if GrandTruth[i][0][0] == Predictions[i]:
        numCorrect += 1
    return numCorrect / samples
```

3 Fit Titanic Data to Custom Neural Network

In this section, we will create two networks to fit the Titanic data to. The first network will contain 2 hidden layers with 10 nodes in each layer, the second network will contain 2 hidden layers with 100 nodes in each layer. Because the data has 4 input features and one out put column, our networks will be 4 by X by X by 1 where X is either 10 or 100 nodes. Each network will use MSE as the loss function, and the sigmoid activation. The MSE results for a few training iterations are shown below. Overall, it looks like the larger network with 100 nodes was more prone to overfitting. So the simple 2x10 network was actually the best in this case. We will take a more in-depth look at the performance of each network in the next step.

[9]:
```
# Define a Neural Network with 2 Hidden Layers that contain 10 Nodes Each
nn1 = Network()
nn1.add(FCLayer(4, 10))
nn1.add(ActivationLayer(sigmoid, sigmoidPrime))
nn1.add(FCLayer(10, 10))
nn1.add(ActivationLayer(sigmoid, sigmoidPrime))
nn1.add(FCLayer(10, 1))
nn1.add(ActivationLayer(sigmoid, sigmoidPrime))
nn1.use(mse, msePrime)

# Train the Neural Network
nn1.fit(xTrainT, yTrainT, epochs=5000, learningRate=0.1)

# Calculate Accuracy
predsTr1 = hardAssignment(nn1.predict(xTrainT))
predsTe1 = hardAssignment(nn1.predict(xTestT))
print('Custom NN (2x10) Train Accuracy: {:.2f}%'.format(100*accuracy(predsTr1,␣
 ↪yTrainT)))
print('Custom NN (2x10) Test  Accuracy: {:.2f}%'.format(100*accuracy(predsTe1,␣
 ↪yTestT)))
```

```
Epoch: 0/5000     Error: 0.236794
Epoch: 1000/5000     Error: 0.126394
Epoch: 2000/5000     Error: 0.121599
Epoch: 3000/5000     Error: 0.118895
Epoch: 4000/5000     Error: 0.116613
Custom NN (2x10) Train Accuracy: 84.24%
Custom NN (2x10) Test  Accuracy: 80.53%
```

[10]:
```
# Define a Neural Network with 2 Hidden Layers that contain 100 Nodes Each
nn2 = Network()
nn2.add(FCLayer(4, 100))
nn2.add(ActivationLayer(sigmoid, sigmoidPrime))
nn2.add(FCLayer(100, 100))
nn2.add(ActivationLayer(sigmoid, sigmoidPrime))
nn2.add(FCLayer(100, 1))
nn2.add(ActivationLayer(sigmoid, sigmoidPrime))
nn2.use(mse, msePrime)

# Train the Neural Network
nn2.fit(xTrainT, yTrainT, epochs=5000, learningRate=0.1)

# Calculate Accuracy
predsTr2 = hardAssignment(nn2.predict(xTrainT))
predsTe2 = hardAssignment(nn2.predict(xTestT))
print('Custom NN (2x100) Train Accuracy: {:.2f}%'.format(100*accuracy(predsTr2,␣
 ↪yTrainT)))
```

```
print('Custom NN (2x100) Test  Accuracy: {:.2f}%'.format(100*accuracy(predsTe2,␣
 ↪yTestT)))
```

```
Epoch: 0/5000     Error: 0.194620
Epoch: 1000/5000     Error: 0.125991
Epoch: 2000/5000     Error: 0.117654
Epoch: 3000/5000     Error: 0.112253
Epoch: 4000/5000     Error: 0.108905
Custom NN (2x100) Train Accuracy: 85.00%
Custom NN (2x100) Test  Accuracy: 77.48%
```

4 Fit Titanic Data to Sklearn Neural Network

As a simple performance test, we will take our custom neural network with 2 hidden layers and 10 nodes (Since that was the best model), and we will compare it with an Sklearn NN that has similar parameters. We set the layer size, activation, solver, and learning rate to match the paraemters in our custom NN, so this should be a fair comparison. After experimenting, we see that Sklearn's NN and the custom built one perform almost neck and neck in regards to test accuracy. Given that our custom model is competitive with the Sklearn model, we will use our custom 2x10 as the primary model for the remainder of this assignment.

[11]:
```
# Create Sklearn NN and Calculate Test Accuracy
clf = MLPClassifier(hidden_layer_sizes=(10,2), activation='logistic',␣
 ↪solver='sgd', learning_rate_init=0.1, random_state=10413641, max_iter=5000).
 ↪fit(xTrain, yTrain)
predsTr = clf.predict(xTrain)
predsTe = clf.predict(xTest)
print('Sklearn NN Train Accuracy: {:.2f}%'.format(100*accuracy_score(predsTr,␣
 ↪yTrain)))
print('Sklearn NN Test  Accuracy: {:.2f}%'.format(100*accuracy_score(predsTe,␣
 ↪yTest)))
```

```
Sklearn NN Train Accuracy: 81.47%
Sklearn NN Test  Accuracy: 79.01%
```

5 Sampling Errors of NN's

Step 3: Print out the performance measures of the full model: - In-sample percent survivors correctly predicted (on training set) - In-sample percent fatalities correctly predicted (on training set) - Out-of-sample percent survivors correctly predicted (on test set) - Out-of-sample percent fatalities correctly predicted (on test set)

In this step, we calculate the in and out-of-sample accuracy rates. After training the model on the training data, we then test the model on the training data again (In-sample) and then on the testing data (Out-of-sample). The results in this section lead to the same conclusions as formulated in the last step, however, the results here offer additional insight into why the models performed in the ways that they did.

[12]:
```
# In-Sample & Out-of-Sample Errors of Sklearn NN
print('----- Performance of Sklearn NN with 2x10 -----')
tn, fp, fn, tp = confusion_matrix(yTrain, predsTr).ravel()
print('Overall In-Sample Accuracy:\t\t\t{:.2f}%'.format(100*accuracy(predsTr,␣
 ↪yTrainT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%\n'.format(100*(tn /␣
 ↪(tn + fp))))

tn, fp, fn, tp = confusion_matrix(yTest, predsTe).ravel()
print('Overall Out-of-Sample Accuracy:\t\t\t{:.2f}%'.
 ↪format(100*accuracy(predsTe, yTestT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%'.format(100*(tn / (tn␣
 ↪+ fp))))
```

```
----- Performance of Sklearn NN with 2x10 -----
Overall In-Sample Accuracy:                     81.47%
Percent Survivors Correctly Predicted:          65.91%
Percent Fatalities Correctly Predicted:         90.94%

Overall Out-of-Sample Accuracy:                 79.01%
Percent Survivors Correctly Predicted:          63.46%
Percent Fatalities Correctly Predicted:         89.24%
```

[13]:
```
# In-Sample & Out-of-Sample Errors of NN1
print('----- Performance of NN1 with 2x10 -----')
tn, fp, fn, tp = confusion_matrix(yTrain, predsTr1).ravel()
print('Overall In-Sample Accuracy:\t\t\t{:.2f}%'.format(100*accuracy(predsTr1,␣
 ↪yTrainT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%\n'.format(100*(tn /␣
 ↪(tn + fp))))

tn, fp, fn, tp = confusion_matrix(yTest, predsTe1).ravel()
print('Overall Out-of-Sample Accuracy:\t\t\t{:.2f}%'.
 ↪format(100*accuracy(predsTe1, yTestT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%'.format(100*(tn / (tn␣
 ↪+ fp))))
```

```
----- Performance of NN1 with 2x10 -----
Overall In-Sample Accuracy:                     84.24%
```

```
Percent Survivors Correctly Predicted:          69.70%
Percent Fatalities Correctly Predicted:         93.09%

Overall Out-of-Sample Accuracy:                 80.53%
Percent Survivors Correctly Predicted:          69.23%
Percent Fatalities Correctly Predicted:         87.97%
```

[14]:
```
# In-Sample & Out-of-Sample Errors of NN2
print('----- Performance of NN2 with 2x100 -----')
tn, fp, fn, tp = confusion_matrix(yTrain, predsTr2).ravel()
print('Overall In-Sample Accuracy:\t\t\t{:.2f}%'.format(100*accuracy(predsTr2,␣
 ↪yTrainT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%\n'.format(100*(tn /␣
 ↪(tn + fp))))

tn, fp, fn, tp = confusion_matrix(yTest, predsTe2).ravel()
print('Overall Out-of-Sample Accuracy:\t\t\t{:.2f}%'.
 ↪format(100*accuracy(predsTe2, yTestT)))
print('Percent Survivors Correctly Predicted:\t\t{:.2f}%'.format(100*(tp / (tp␣
 ↪+ fn))))
print('Percent Fatalities Correctly Predicted:\t\t{:.2f}%'.format(100*(tn / (tn␣
 ↪+ fp))))
```

```
----- Performance of NN2 with 2x100 -----
Overall In-Sample Accuracy:                     85.00%
Percent Survivors Correctly Predicted:          72.73%
Percent Fatalities Correctly Predicted:         92.47%

Overall Out-of-Sample Accuracy:                 77.48%
Percent Survivors Correctly Predicted:          66.35%
Percent Fatalities Correctly Predicted:         84.81%
```

6 Model Comparison

Step 4: Compare the in-sample and out-of-sample accuracy with the pruned decision tree obtained in Homework #3.

In this step, we compare the in and out-of-sample accuracies for each of the models with the pruned decision tree from the previous homework. The table below shows the accuracy percentage for each of the models. Overall, it looks like the custom 2x10 NN that I built performs the best out of all the models. The pruned decison tree was very close in terms of testing accuracy as well.

[15]:
```
# Model Comparison
nn1AccTr = round(100*accuracy(predsTr1, yTrainT),2)
nn1AccTe = round(100*accuracy(predsTe1, yTestT ),2)
```

```
nn2AccTr = round(100*accuracy(predsTr2, yTrainT),2)
nn2AccTe = round(100*accuracy(predsTe2, yTestT ),2)
SklAccTr = round(100*accuracy(predsTr,  yTrainT),2)
SklAccTe = round(100*accuracy(predsTe,  yTestT ),2)
dtrAccTr = 81.09
dtrAccTe = 79.39

fig = go.Figure(data=[go.Table(header=dict(values=['Model', 'In-Sample Accuracy␣
 ↪(%)', 'Out-of-Sample Accuracy (%)']),
                 cells=dict(values=[['Custom 2x10 NN', 'Custom 2x100 NN',␣
 ↪'Sklearn 2x10 NN', 'Pruned DT'],
                                    [nn1AccTr, nn2AccTr, SklAccTr, dtrAccTr],
                                    [nn1AccTe, nn2AccTe, SklAccTe, dtrAccTe]]))])
fig.show()
```

Model	In-Sample Accuracy (%)	Out-of-Sample Accuracy (%)
Custom 2x10 NN	84.24	80.53
Custom 2x100 NN	85	77.48
Sklearn 2x10 NN	81.47	79.01
Pruned DT	81.09	79.39