Logistic Regression, MLP and Random Forest with Sklearn and Feed Forward Neural Net with PyTorch on Baxter Dataset

Let's make sure we are using the Python version we want

```
import sys
print(sys.version)
```

```
[Output] 3.6.6 (v3.6.6:4cf1f54eb7, Jun 26 2018, 17:02:57)
[Output] [GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
```

Let's make sure we are in our project directory

```
import os
print(os.getcwd())
```

```
[Output] /Users/Begum/Documents/DeepLearning
```

Start by importing modules that will be necessary

```
###### IMPORT MODULES #######
# We need to use a backend for matplotlib
# https://stackoverflow.com/questions/21784641/installation-issue-with-matplotlib-python/21789908#21789908
import matplotlib as mpl
mpl.use('TkAgg')
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold, cross_val_score, validation_curve
from sklearn import linear_model
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

```
from sympy import *
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier
from scipy import interp
from itertools import cycle
from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import StratifiedKFold
# dependencies for statistic analysis
from scipy import stats
#importing our parameter tuning dependencies
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import (cross_val_score, GridSearchCV, StratifiedKFold, ShuffleSplit )
#importing our dependencies for Feature Selection
from sklearn.feature_selection import (SelectKBest, chi2, RFE, RFECV)
from sklearn.linear_model import LogisticRegression, RandomizedLogisticRegression
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import f1_score
# Importing our sklearn dependencies for the modeling
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import (accuracy_score, confusion_matrix, classification_report, roc_curve, auc)
from sklearn.neural_network import MLPClassifier
```

```
from itertools import cycle
from scipy import interp
import warnings
from sklearn.model_selection import StratifiedKFold
import torch
from torch.autograd import Variable
import torch.utils.data as data_utils
import torch.nn.init as init
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
warnings.filterwarnings('ignore')
```

Read and prepare data

```
## Read in the data
shared = pd.read_table("data/baxter.0.03.subsample.shared")
shared.head()
meta = pd.read_table("data/metadata.tsv")
## Check and visualize the data
print(meta.head())
```

```
[Output]     sample  fit_result        Site    ...    Diabetes_Med stage  Location
[Output] 0  2003650           0  U Michigan    ...               0     0       NaN
[Output] 1  2005650           0  U Michigan    ...               0     0       NaN
[Output] 2  2007660          26  U Michigan    ...               0     0       NaN
[Output] 3  2009650          10     Toronto    ...               0     0       NaN
[Output] 4  2013660           0  U Michigan    ...               0     0       NaN
[Output]
[Output] [5 rows x 27 columns]
```

```
print(shared.head())
## Remove unnecessary columns from meta
```

```
[Output]    label    Group  numOtus    ...     Otu11278  Otu11280  Otu11281
[Output] 0   0.03  2003650     6920    ...            0         0         0
[Output] 1   0.03  2005650     6920    ...            0         0         0
[Output] 2   0.03  2007660     6920    ...            0         0         0
[Output] 3   0.03  2009650     6920    ...            0         0         0
[Output] 4   0.03  2013660     6920    ...            0         0         0
[Output]
[Output] [5 rows x 6923 columns]
```

```
meta = meta[['sample','dx']]
## Rename the column name "Group" to match the "sample" in meta
shared = shared.rename(index=str, columns={"Group":"sample"})
## Merge the 2 datasets on sample
data=pd.merge(meta,shared,on=['sample'])
## Remove adenoma samples
data= data[data.dx.str.contains("adenoma") == False]
## Drop all except OTU columns for x
x = data.drop(["sample", "dx", "numOtus", "label"], axis=1)
## Cancer =1 Normal =0
diagnosis = { "cancer":1, "normal":0}
##Generate y which only has diagnosis as 0 and 1
y = data["dx"].replace(diagnosis)
# y = np.eye(2, dtype='uint8')[y]
## Drop if NA elements
y.dropna()
x.dropna()
```

```
print(x.head())
```

```
[Output]     Otu00001  Otu00002  Otu00003    ...      Otu11278  Otu11280  Otu11281
[Output] 0        350       268       213    ...             0         0         0
[Output] 1        568      1320        13    ...             0         0         0
[Output] 2        151       756       802    ...             0         0         0
[Output] 4       1409       174         0    ...             0         0         0
[Output] 5        167       712       213    ...             0         0         0
[Output]
[Output] [5 rows x 6920 columns]
```

```
print(y.head())
```

```
[Output] 0    0
[Output] 1    0
[Output] 2    0
[Output] 4    0
[Output] 5    0
[Output] Name: dx, dtype: int64
```

```
print(len(x))
```

```
[Output] 292
```

```
print(len(y))
```

```
[Output] 292
```

Split the data to generate training and testing set %80-20

Here we also want to shuffle the data and set a random state

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=82089, shuffle=True)
```

Now let's define a L2 regularized logistic regression model

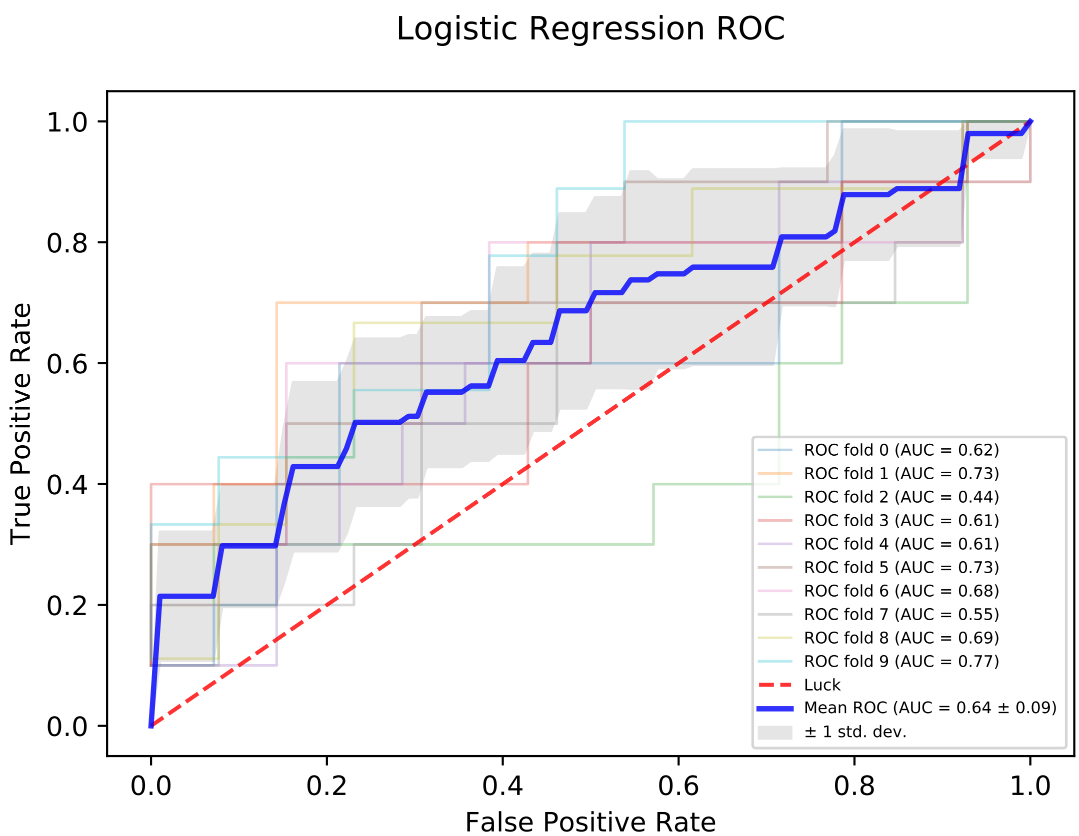
```
## Define L2 regularized logistic classifier
logreg = linear_model.LogisticRegression(C=0.01)
```

Plot ROC curve for Logistic Regression training set

We split the training set to 10 to cross-validate

```
cv = StratifiedKFold(n_splits=10)
tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
## Convert from pandas dataframe to numpy
X=x_train.values
Y=y_train.values
# Plot the ROC curve over 10 iterations for each split
#logistic_plot = plt.figure()
i = 0
for train, test in cv.split(X,Y):
    probas_ = logreg.fit(X[train], Y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(Y[test], probas_[:, 1])
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)
    plt.plot(fpr, tpr, lw=1, alpha=0.3, label='ROC fold %d (AUC = %0.2f)' % (i, roc_auc))
    i += 1
```

```
plt.plot([0, 1], [0, 1], linestyle='--', color='r', label='Luck', alpha=.8)
mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)
plt.plot(mean_fpr, mean_tpr, color='b', label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc), lw=2,
std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2, label=r'$\pm$ 1 std. dev.')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC\n')
plt.legend(loc="lower right", fontsize=6)
plt.show()
#logistic_plot.savefig('results/figures/Logit_Baxter.png', dpi=1000)
```



Predict using the Logistic Regression model on the test set

```
y_pred = logreg.predict(x_test)
print('Accuracy of logistic regression classifier on test set: {:.2f}'.format(logreg.score(x_test, y_test)*100),
```

```
[Output] Accuracy of logistic regression classifier on test set: 67.80 %
```

Now let's define a Multi-layer Perceptron Neural Network Model

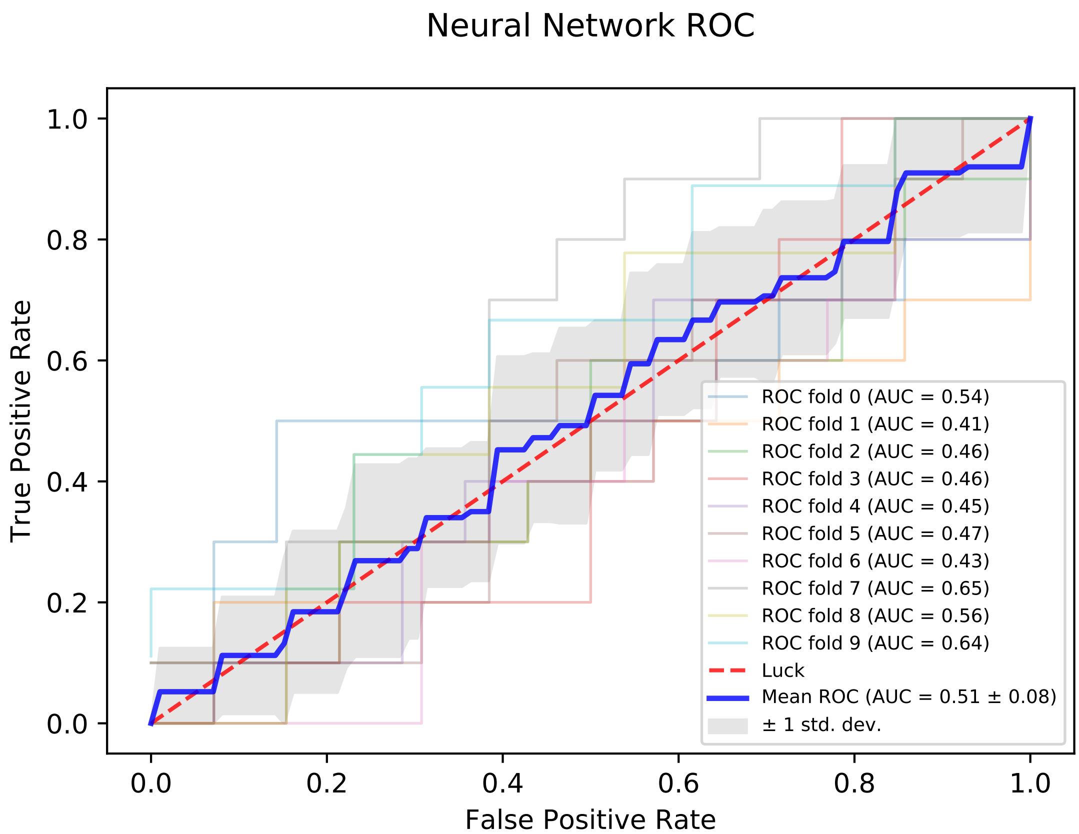
```
clf = MLPClassifier(activation='logistic', alpha=0.001, batch_size='auto',
       beta_1=0.9, beta_2=0.999, early_stopping=True, epsilon=1e-08,
       hidden_layer_sizes=(100,), learning_rate='adaptive',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=1, shuffle=True,
```

```
       solver='sgd', tol=0.0001, validation_fraction=0.1, verbose=False,
       warm_start=False)
```

Plot ROC curve for MLP on training set

We split the training set to 10 to cross-validate

```
cv = StratifiedKFold(n_splits=10)
tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
mlp_plot = plt.figure()
i = 0
for train, test in cv.split(X,Y):
    probas_ = clf.fit(X[train], Y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
    fpr, tpr, thresholds = roc_curve(Y[test], probas_[:, 1])
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)
    plt.plot(fpr, tpr, lw=1, alpha=0.3, label='ROC fold %d (AUC = %0.2f)' % (i, roc_auc))
    i += 1
plt.plot([0, 1], [0, 1], linestyle='--', color='r', label='Luck', alpha=.8)
mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)
plt.plot(mean_fpr, mean_tpr, color='b', label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc), lw=2,
std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2, label=r'$\pm$ 1 std. dev.')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Neural Network ROC\n')
plt.legend(loc="lower right", fontsize=7)
plt.show()
#mlp_plot.savefig('results/figures/MLP_Baxter.png', dpi=1000)
```



Predict using the MLP model on the test set

```
y_pred = clf.predict(x_test)
## Print accuracy
print("Performance Accuracy on the Testing data:", round(clf.score(x_test, y_test) *100))
```

```
[Output] Performance Accuracy on the Testing data: 66.0
```

```
print("Number of correct classifiers:", round(accuracy_score(y_test, y_pred, normalize=False)))
```

```
[Output] Number of correct classifiers: 39
```

Now let's define a Random Forest model

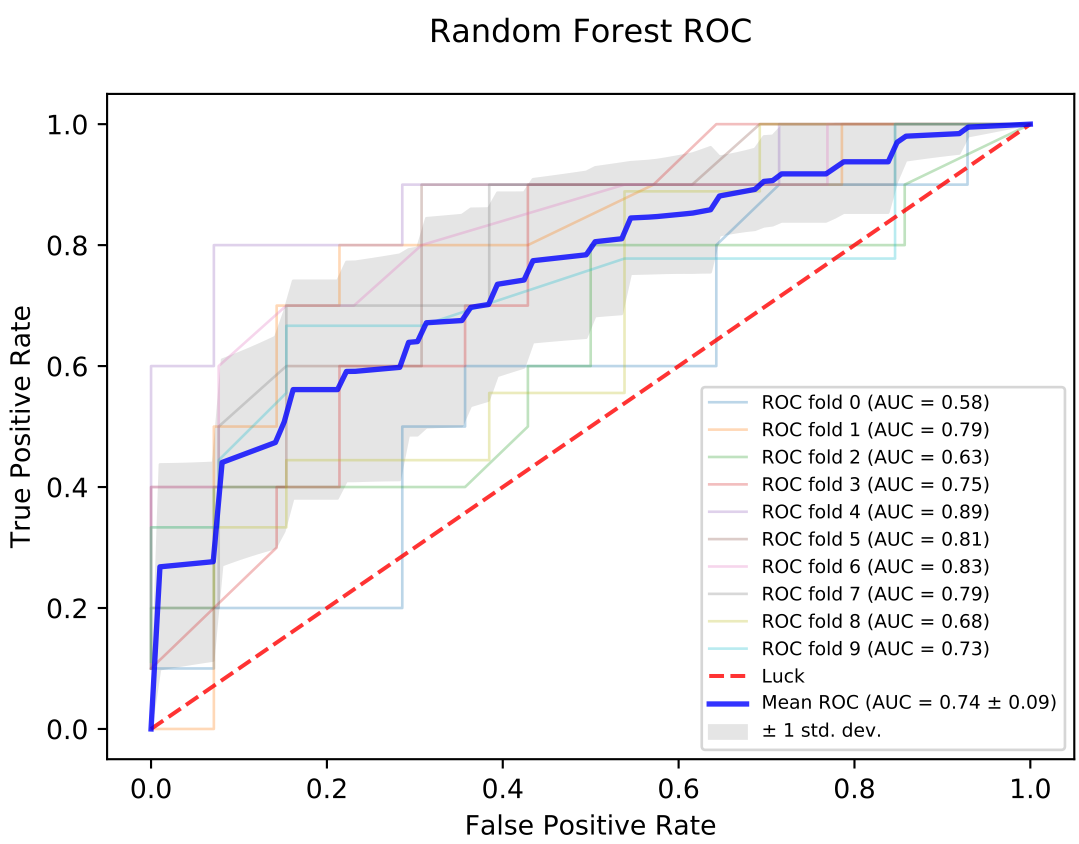
```
rfc = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=150, n_jobs=1, oob_score=True, random_state=None,
            verbose=0, warm_start=False)
```

Plot ROC curve for Random Forest on training set

We split the training set to 10 to cross-validate

```
RF_plot = plt.figure()
cv = StratifiedKFold(n_splits=10)
tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
i = 0
for train, test in cv.split(X,Y):
    probas_ = rfc.fit(X[train], Y[train]).predict_proba(X[test])
    # Compute ROC curve and area the curve
```

```
    fpr, tpr, thresholds = roc_curve(Y[test], probas_[:, 1])
    tprs.append(interp(mean_fpr, fpr, tpr))
    tprs[-1][0] = 0.0
    roc_auc = auc(fpr, tpr)
    aucs.append(roc_auc)
    plt.plot(fpr, tpr, lw=1, alpha=0.3, label='ROC fold %d (AUC = %0.2f)' % (i, roc_auc))
    i += 1
plt.plot([0, 1], [0, 1], linestyle='--', color='r', label='Luck', alpha=.8)
mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)
plt.plot(mean_fpr, mean_tpr, color='b', label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc), lw=2,
std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2, label=r'$\pm$ 1 std. dev.')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Random Forest ROC\n')
plt.legend(loc="lower right", fontsize=7)
plt.show()
#RF_plot.savefig('results/figures/Random_Forest_Baxter.png', dpi=1000)
```



Predict using the Random Forest model on the test set

```
y_pred = rfc.predict(x_test)
print("Performance Accuracy on the Testing data:", round(rfc.score(x_test, y_test) *100))
```

```
[Output] Performance Accuracy on the Testing data: 71.0
```

```
print("Number of correct classifiers:", round(accuracy_score(y_test, y_pred, normalize=False)))
```

```
[Output] Number of correct classifiers: 42
```

Now let's define a 1 hidden layer(n=100) Feed Forward Neural Net

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(6920, 100)
        self.fc2 = nn.Linear(100, 2)
    def forward(self, x):
        x = self.fc1(x)
        x = F.dropout(x, p=0.1)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.sigmoid(x)
        return x
net = Net()
## Batch size allows for random sampling of the dataset during training
batch_size = 50
num_epochs = 50
learning_rate = 0.0001
batch_no = len(x_train) // batch_size
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
from sklearn.utils import shuffle
from torch.autograd import Variable
from scipy import interp
from sklearn.metrics import (accuracy_score, confusion_matrix, classification_report, roc_curve, auc)
## ROC plot for training
pyTorch_plot = plt.figure()
tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
for epoch in range(num_epochs):
    x_train, y_train = shuffle(x_train, y_train)
    # Mini batch learning
    for i in range(batch_no):
        start = i * batch_size
        end = start + batch_size
        x_var = Variable(torch.FloatTensor(x_train.values[start:end]))
        y_var = Variable(torch.LongTensor(y_train.values[start:end]))
        # Forward + Backward + Optimize
        ypred_var = net(x_var)
        loss =criterion(ypred_var, y_var)
        correct_num = 0
        ## The outputs of the model (ypred_var) are energies for the 10 classes.
        #Higher the energy for a class, the more the network thinks that the image is of the         #particular
        values, labels = torch.max(ypred_var, 1)
        correct_num = np.sum(labels.data.numpy() == y_var.numpy())
        fpr, tpr, thresholds = roc_curve(y_var.numpy(), labels.data.numpy())
        tprs.append(interp(mean_fpr, fpr, tpr))
        tprs[-1][0] = 0.0
        roc_auc = auc(fpr, tpr)
        aucs.append(roc_auc)
        #plt.plot(fpr, tpr, lw=1, alpha=0.3, label='ROC fold %d (AUC = %0.2f)' % (epoch, roc_auc))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print('Epoch [%d], Loss:%.4f, Accuracy:%.4f' % (epoch, loss.data[0], correct_num/len(labels)))
```

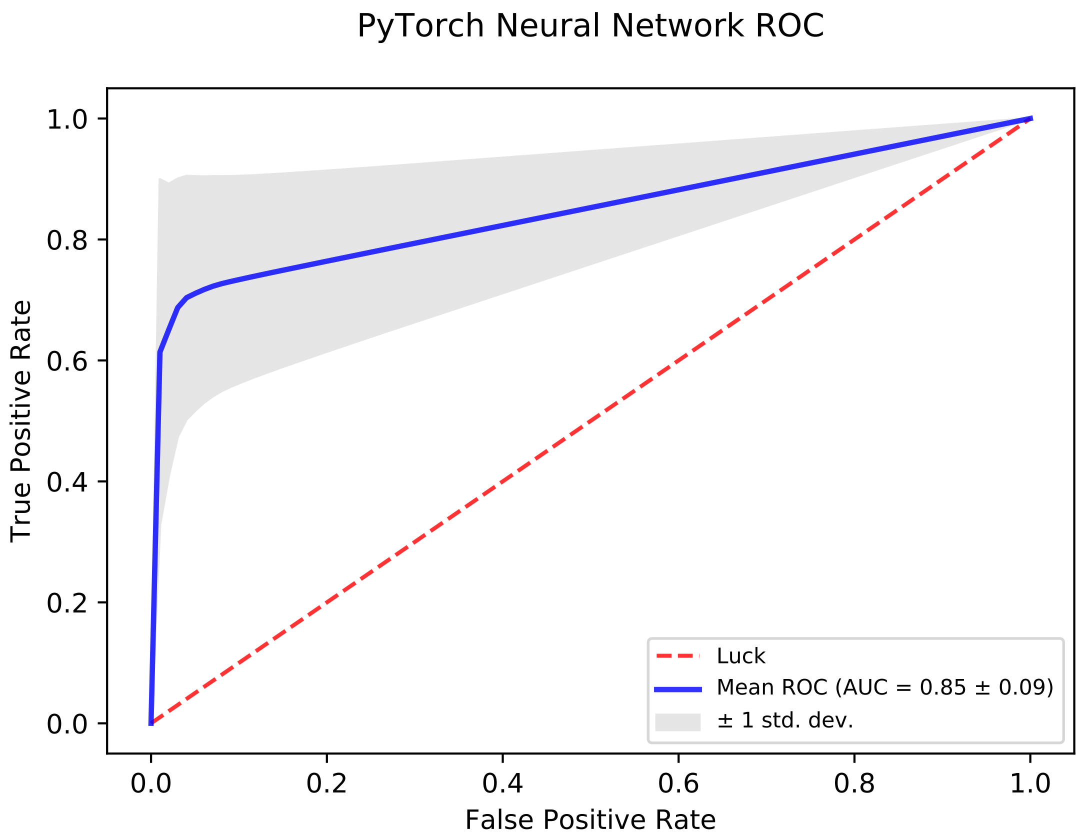
```
[Output] Epoch [0], Loss:0.6687, Accuracy:0.5800
[Output] Epoch [0], Loss:0.7109, Accuracy:0.5800
[Output] Epoch [0], Loss:0.6285, Accuracy:0.6000
[Output] Epoch [0], Loss:0.6118, Accuracy:0.6800
[Output] Epoch [1], Loss:0.6656, Accuracy:0.5200
[Output] Epoch [1], Loss:0.5541, Accuracy:0.7200
[Output] Epoch [1], Loss:0.6389, Accuracy:0.6400
[Output] Epoch [1], Loss:0.5498, Accuracy:0.6800
[Output] Epoch [2], Loss:0.5691, Accuracy:0.6600
[Output] Epoch [2], Loss:0.5994, Accuracy:0.5800
[Output] Epoch [2], Loss:0.5055, Accuracy:0.7800
[Output] Epoch [2], Loss:0.5497, Accuracy:0.6600
[Output] Epoch [3], Loss:0.5455, Accuracy:0.7200
[Output] Epoch [3], Loss:0.5140, Accuracy:0.7800
[Output] Epoch [3], Loss:0.5426, Accuracy:0.7600
[Output] Epoch [3], Loss:0.5778, Accuracy:0.6400
[Output] Epoch [4], Loss:0.5070, Accuracy:0.7400
[Output] Epoch [4], Loss:0.5756, Accuracy:0.6000
[Output] Epoch [4], Loss:0.5530, Accuracy:0.7400
[Output] Epoch [4], Loss:0.4730, Accuracy:0.8200
[Output] Epoch [5], Loss:0.5106, Accuracy:0.7200
[Output] Epoch [5], Loss:0.5217, Accuracy:0.7400
[Output] Epoch [5], Loss:0.5019, Accuracy:0.7400
[Output] Epoch [5], Loss:0.5093, Accuracy:0.8200
[Output] Epoch [6], Loss:0.5104, Accuracy:0.8000
[Output] Epoch [6], Loss:0.4703, Accuracy:0.8000
[Output] Epoch [6], Loss:0.5285, Accuracy:0.6800
[Output] Epoch [6], Loss:0.4618, Accuracy:0.8800
[Output] Epoch [7], Loss:0.4591, Accuracy:0.8400
[Output] Epoch [7], Loss:0.5344, Accuracy:0.8000
[Output] Epoch [7], Loss:0.4308, Accuracy:0.8200
[Output] Epoch [7], Loss:0.5124, Accuracy:0.8200
[Output] Epoch [8], Loss:0.4489, Accuracy:0.8400
[Output] Epoch [8], Loss:0.4621, Accuracy:0.8600
[Output] Epoch [8], Loss:0.4659, Accuracy:0.8400
[Output] Epoch [8], Loss:0.4803, Accuracy:0.8000
[Output] Epoch [9], Loss:0.4931, Accuracy:0.8200
[Output] Epoch [9], Loss:0.4520, Accuracy:0.8000
[Output] Epoch [9], Loss:0.4885, Accuracy:0.8000
[Output] Epoch [9], Loss:0.4105, Accuracy:0.8800
[Output] Epoch [10], Loss:0.4525, Accuracy:0.8200
[Output] Epoch [10], Loss:0.4500, Accuracy:0.7800
[Output] Epoch [10], Loss:0.4421, Accuracy:0.9000
[Output] Epoch [10], Loss:0.4513, Accuracy:0.8600
[Output] Epoch [11], Loss:0.4686, Accuracy:0.8000
[Output] Epoch [11], Loss:0.4656, Accuracy:0.8400
[Output] Epoch [11], Loss:0.4043, Accuracy:0.9000
[Output] Epoch [11], Loss:0.3812, Accuracy:0.9400
[Output] Epoch [12], Loss:0.3914, Accuracy:0.9400
[Output] Epoch [12], Loss:0.4509, Accuracy:0.8600
[Output] Epoch [12], Loss:0.4286, Accuracy:0.8400
[Output] Epoch [12], Loss:0.4723, Accuracy:0.7800
[Output] Epoch [13], Loss:0.4112, Accuracy:0.9000
[Output] Epoch [13], Loss:0.4518, Accuracy:0.7800
[Output] Epoch [13], Loss:0.4043, Accuracy:0.9000
[Output] Epoch [13], Loss:0.4062, Accuracy:0.8600
[Output] Epoch [14], Loss:0.3574, Accuracy:0.9400
[Output] Epoch [14], Loss:0.4441, Accuracy:0.8200
[Output] Epoch [14], Loss:0.4526, Accuracy:0.8400
[Output] Epoch [14], Loss:0.3676, Accuracy:0.9400
[Output] Epoch [15], Loss:0.4127, Accuracy:0.8600
```

```
[Output] Epoch [15], Loss:0.4235, Accuracy:0.8200
[Output] Epoch [15], Loss:0.3878, Accuracy:0.9400
[Output] Epoch [15], Loss:0.3845, Accuracy:0.9400
[Output] Epoch [16], Loss:0.3692, Accuracy:0.9400
[Output] Epoch [16], Loss:0.4438, Accuracy:0.8200
[Output] Epoch [16], Loss:0.3826, Accuracy:0.9000
[Output] Epoch [16], Loss:0.3950, Accuracy:0.8800
[Output] Epoch [17], Loss:0.4333, Accuracy:0.8200
[Output] Epoch [17], Loss:0.4135, Accuracy:0.8200
[Output] Epoch [17], Loss:0.3680, Accuracy:0.9600
[Output] Epoch [17], Loss:0.3989, Accuracy:0.9000
[Output] Epoch [18], Loss:0.4409, Accuracy:0.7800
[Output] Epoch [18], Loss:0.3631, Accuracy:0.9600
[Output] Epoch [18], Loss:0.3769, Accuracy:0.9200
[Output] Epoch [18], Loss:0.4062, Accuracy:0.8600
[Output] Epoch [19], Loss:0.3825, Accuracy:0.9200
[Output] Epoch [19], Loss:0.3568, Accuracy:0.9400
[Output] Epoch [19], Loss:0.4035, Accuracy:0.8600
[Output] Epoch [19], Loss:0.4096, Accuracy:0.8600
[Output] Epoch [20], Loss:0.3692, Accuracy:0.9600
[Output] Epoch [20], Loss:0.3594, Accuracy:0.9400
[Output] Epoch [20], Loss:0.3773, Accuracy:0.8600
[Output] Epoch [20], Loss:0.3903, Accuracy:0.8800
[Output] Epoch [21], Loss:0.3895, Accuracy:0.8800
[Output] Epoch [21], Loss:0.3462, Accuracy:0.9600
[Output] Epoch [21], Loss:0.3814, Accuracy:0.9200
[Output] Epoch [21], Loss:0.4237, Accuracy:0.8600
[Output] Epoch [22], Loss:0.3889, Accuracy:0.8600
[Output] Epoch [22], Loss:0.3539, Accuracy:0.9400
[Output] Epoch [22], Loss:0.3731, Accuracy:0.9200
[Output] Epoch [22], Loss:0.3934, Accuracy:0.9400
[Output] Epoch [23], Loss:0.3538, Accuracy:0.9400
[Output] Epoch [23], Loss:0.3458, Accuracy:0.9400
[Output] Epoch [23], Loss:0.3929, Accuracy:0.9200
[Output] Epoch [23], Loss:0.4255, Accuracy:0.8200
[Output] Epoch [24], Loss:0.3527, Accuracy:0.9600
[Output] Epoch [24], Loss:0.3728, Accuracy:0.9000
[Output] Epoch [24], Loss:0.3608, Accuracy:0.9200
[Output] Epoch [24], Loss:0.3958, Accuracy:0.8800
[Output] Epoch [25], Loss:0.3879, Accuracy:0.9200
[Output] Epoch [25], Loss:0.3607, Accuracy:0.9000
[Output] Epoch [25], Loss:0.4029, Accuracy:0.8600
[Output] Epoch [25], Loss:0.3881, Accuracy:0.9000
[Output] Epoch [26], Loss:0.3599, Accuracy:0.9200
[Output] Epoch [26], Loss:0.3802, Accuracy:0.9000
[Output] Epoch [26], Loss:0.3801, Accuracy:0.8600
[Output] Epoch [26], Loss:0.4025, Accuracy:0.9200
[Output] Epoch [27], Loss:0.4078, Accuracy:0.8400
[Output] Epoch [27], Loss:0.3868, Accuracy:0.9200
[Output] Epoch [27], Loss:0.3645, Accuracy:0.9400
[Output] Epoch [27], Loss:0.3675, Accuracy:0.9000
[Output] Epoch [28], Loss:0.3599, Accuracy:0.9600
[Output] Epoch [28], Loss:0.3452, Accuracy:0.9200
[Output] Epoch [28], Loss:0.3526, Accuracy:0.9200
[Output] Epoch [28], Loss:0.4180, Accuracy:0.8600
[Output] Epoch [29], Loss:0.3449, Accuracy:0.9400
[Output] Epoch [29], Loss:0.4028, Accuracy:0.8600
[Output] Epoch [29], Loss:0.3933, Accuracy:0.9400
[Output] Epoch [29], Loss:0.3828, Accuracy:0.9200
[Output] Epoch [30], Loss:0.3750, Accuracy:0.9000
[Output] Epoch [30], Loss:0.4066, Accuracy:0.8600
```

```
[Output] Epoch [30], Loss:0.3518, Accuracy:0.9600
[Output] Epoch [30], Loss:0.3960, Accuracy:0.9000
[Output] Epoch [31], Loss:0.3456, Accuracy:0.9600
[Output] Epoch [31], Loss:0.3778, Accuracy:0.9000
[Output] Epoch [31], Loss:0.3775, Accuracy:0.9200
[Output] Epoch [31], Loss:0.3843, Accuracy:0.8800
[Output] Epoch [32], Loss:0.3893, Accuracy:0.9200
[Output] Epoch [32], Loss:0.3542, Accuracy:0.9200
[Output] Epoch [32], Loss:0.3827, Accuracy:0.8800
[Output] Epoch [32], Loss:0.3600, Accuracy:0.9400
[Output] Epoch [33], Loss:0.3539, Accuracy:0.9400
[Output] Epoch [33], Loss:0.3731, Accuracy:0.9200
[Output] Epoch [33], Loss:0.3603, Accuracy:0.9200
[Output] Epoch [33], Loss:0.3710, Accuracy:0.9000
[Output] Epoch [34], Loss:0.3750, Accuracy:0.9200
[Output] Epoch [34], Loss:0.3679, Accuracy:0.9000
[Output] Epoch [34], Loss:0.3708, Accuracy:0.9200
[Output] Epoch [34], Loss:0.3447, Accuracy:0.9400
[Output] Epoch [35], Loss:0.3755, Accuracy:0.9000
[Output] Epoch [35], Loss:0.3466, Accuracy:0.9600
[Output] Epoch [35], Loss:0.3877, Accuracy:0.9000
[Output] Epoch [35], Loss:0.3695, Accuracy:0.8800
[Output] Epoch [36], Loss:0.3600, Accuracy:0.9200
[Output] Epoch [36], Loss:0.3716, Accuracy:0.8800
[Output] Epoch [36], Loss:0.3753, Accuracy:0.8800
[Output] Epoch [36], Loss:0.3733, Accuracy:0.9600
[Output] Epoch [37], Loss:0.3528, Accuracy:0.9000
[Output] Epoch [37], Loss:0.3654, Accuracy:0.9600
[Output] Epoch [37], Loss:0.3602, Accuracy:0.9400
[Output] Epoch [37], Loss:0.3829, Accuracy:0.9200
[Output] Epoch [38], Loss:0.3527, Accuracy:0.9400
[Output] Epoch [38], Loss:0.3598, Accuracy:0.9400
[Output] Epoch [38], Loss:0.3802, Accuracy:0.9400
[Output] Epoch [38], Loss:0.3600, Accuracy:0.9000
[Output] Epoch [39], Loss:0.3600, Accuracy:0.9200
[Output] Epoch [39], Loss:0.3676, Accuracy:0.9000
[Output] Epoch [39], Loss:0.3373, Accuracy:0.9600
[Output] Epoch [39], Loss:0.3724, Accuracy:0.9400
[Output] Epoch [40], Loss:0.3676, Accuracy:0.9000
[Output] Epoch [40], Loss:0.3724, Accuracy:0.9200
[Output] Epoch [40], Loss:0.3676, Accuracy:0.9000
[Output] Epoch [40], Loss:0.3673, Accuracy:0.9200
[Output] Epoch [41], Loss:0.3794, Accuracy:0.9400
[Output] Epoch [41], Loss:0.3822, Accuracy:0.8800
[Output] Epoch [41], Loss:0.3521, Accuracy:0.9400
[Output] Epoch [41], Loss:0.3604, Accuracy:0.9000
[Output] Epoch [42], Loss:0.3673, Accuracy:0.9000
[Output] Epoch [42], Loss:0.3522, Accuracy:0.9400
[Output] Epoch [42], Loss:0.3667, Accuracy:0.9200
[Output] Epoch [42], Loss:0.3863, Accuracy:0.9000
[Output] Epoch [43], Loss:0.3371, Accuracy:0.9400
[Output] Epoch [43], Loss:0.3796, Accuracy:0.8800
[Output] Epoch [43], Loss:0.3479, Accuracy:0.9600
[Output] Epoch [43], Loss:0.4022, Accuracy:0.8600
[Output] Epoch [44], Loss:0.3600, Accuracy:0.9000
[Output] Epoch [44], Loss:0.3449, Accuracy:0.9400
[Output] Epoch [44], Loss:0.3676, Accuracy:0.9200
[Output] Epoch [44], Loss:0.3649, Accuracy:0.9200
[Output] Epoch [45], Loss:0.3676, Accuracy:0.8800
[Output] Epoch [45], Loss:0.3432, Accuracy:0.9600
[Output] Epoch [45], Loss:0.3528, Accuracy:0.9200
```

```
[Output] Epoch [45], Loss:0.3795, Accuracy:0.9400
[Output] Epoch [46], Loss:0.3651, Accuracy:0.9000
[Output] Epoch [46], Loss:0.3522, Accuracy:0.9200
[Output] Epoch [46], Loss:0.3597, Accuracy:0.9200
[Output] Epoch [46], Loss:0.3706, Accuracy:0.9200
[Output] Epoch [47], Loss:0.3492, Accuracy:0.9400
[Output] Epoch [47], Loss:0.3660, Accuracy:0.9000
[Output] Epoch [47], Loss:0.3525, Accuracy:0.9800
[Output] Epoch [47], Loss:0.3598, Accuracy:0.9000
[Output] Epoch [48], Loss:0.3758, Accuracy:0.9000
[Output] Epoch [48], Loss:0.3570, Accuracy:0.9600
[Output] Epoch [48], Loss:0.3673, Accuracy:0.8800
[Output] Epoch [48], Loss:0.3372, Accuracy:0.9400
[Output] Epoch [49], Loss:0.3495, Accuracy:0.9400
[Output] Epoch [49], Loss:0.3602, Accuracy:0.9200
[Output] Epoch [49], Loss:0.3520, Accuracy:0.9400
[Output] Epoch [49], Loss:0.3673, Accuracy:0.8800
```

```
plt.plot([0, 1], [0, 1], linestyle='--', color='r', label='Luck', alpha=.8)
mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)
plt.plot(mean_fpr, mean_tpr, color='b', label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc), lw=2,
std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2, label=r'$\pm$ 1 std. dev.')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('PyTorch Neural Network ROC\n')
plt.legend(loc="lower right", fontsize=8)
plt.show()
#pyTorch_plot.savefig('results/figures/pyTorch_Baxter.png', dpi=1000)
# Evaluate the model on test set
```




```
net.eval()
pred = net(torch.from_numpy(x_test.values).float())
pred = torch.max(pred,1)[1]
len(pred)
pred = pred.data.numpy()
print(accuracy_score(y_test, pred))
```

```
[Output] 0.6101694915254238
```

```
print(confusion_matrix(y_test, pred))
```

```
[Output] [[26 11]
[Output]  [12 10]]
```