In [150]:
```python
import numpy as np
import pandas as pd
import seaborn as sns
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

import sklearn as skl
import skorch as skr
```

In [155]:
```python
print('pands', pd.__version__)
print('numpy', np.__version__)
print('seaborn',sns.__version__)
print('torch',torch.__version__)
print('skorch',skr.__version__)
```

```
pands 1.4.2
numpy 1.21.5
seaborn 0.11.2
torch 1.13.1+cpu
skorch 0.12.1
```

In [2]:
```python
df = pd.read_excel('ctgdata.xlsx')
df.head()
```

Out[2]:

	Unnamed: 0	b	e	LBE	LB	AC	FM	UC	ASTV	MSTV	...	C	D	E	AD	DE	LD	FS	SUSP	CLASS	NSP
0	0	240	357	120	120	0	0	0	73	0.5	...	0	0	0	0	0	0	1	0	9	2
1	1	5	632	132	132	4	0	4	17	2.1	...	0	0	0	1	0	0	0	0	6	1
2	2	177	779	133	133	2	0	5	16	2.1	...	0	0	0	1	0	0	0	0	6	1
3	3	411	1192	134	134	2	0	6	16	2.4	...	0	0	0	1	0	0	0	0	6	1
4	4	533	1147	132	132	4	0	5	16	2.4	...	0	0	0	0	0	0	0	0	2	1

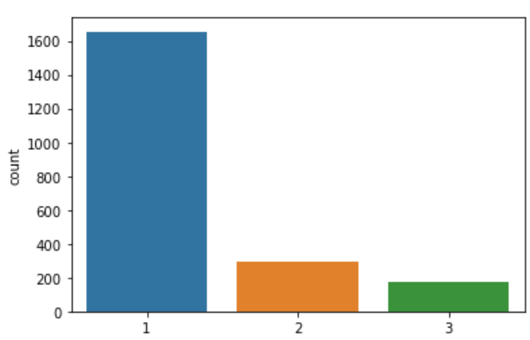
5 rows × 37 columns

Data exploration

I know from the initial data exploration done during the group coursework that this dataset has been preprocessed. However during the group cw we decided against any feature extraction/transfromation because it was not necessary. Some research has suggested feature extraction increases accuracy. Thus, features with low correlations and vriances will be removed.

In [3]:
```python
#class distribution
sns.countplot(x = 'NSP', data=df)
```

Out[3]: <AxesSubplot:xlabel='NSP', ylabel='count'>



In [5]:
```
, 1ALTV', 'MLTV', 'Width', 'Min', 'Max', 'Nmax', 'Nzeros', 'Mode', 'Mean', 'Median', 'Variance', 'Tendency', 'SUSP', 'CLASS
```

UCI says there are only 23 attributes, but this shows 35. According to UCI feature are: LB, AC, FM, UC, DL, DS, DP, ASTV, MSTV, ALTV, MLTV, Width, Min, Max, NMax, Nzeros, Mode, Mean, Median, Variance, Tendency, Class, NSP.I'll keep SUSP because it appears to have a stornger corr than most. All others will be dropped

In [4]:
```
#reduced to 23 attributes.
df.head()
```

Out[4]:

	Unnamed: 0	b	e	LBE	LB	AC	FM	UC	ASTV	MSTV	...	C	D	E	AD	DE	LD	FS	SUSP	CLASS	NSP
0	0	240	357	120	120	0	0	0	73	0.5	...	0	0	0	0	0	0	1	0	9	2
1	1	5	632	132	132	4	0	4	17	2.1	...	0	0	0	1	0	0	0	0	6	1
2	2	177	779	133	133	2	0	5	16	2.1	...	0	0	0	1	0	0	0	0	6	1
3	3	411	1192	134	134	2	0	6	16	2.4	...	0	0	0	1	0	0	0	0	6	1
4	4	533	1147	132	132	4	0	5	16	2.4	...	0	0	0	0	0	0	0	0	2	1

5 rows × 37 columns

the data is imbalanced, which was known. SMOTE did little to improve this during prelimbary scikit phase. Will feature extraction improve? /

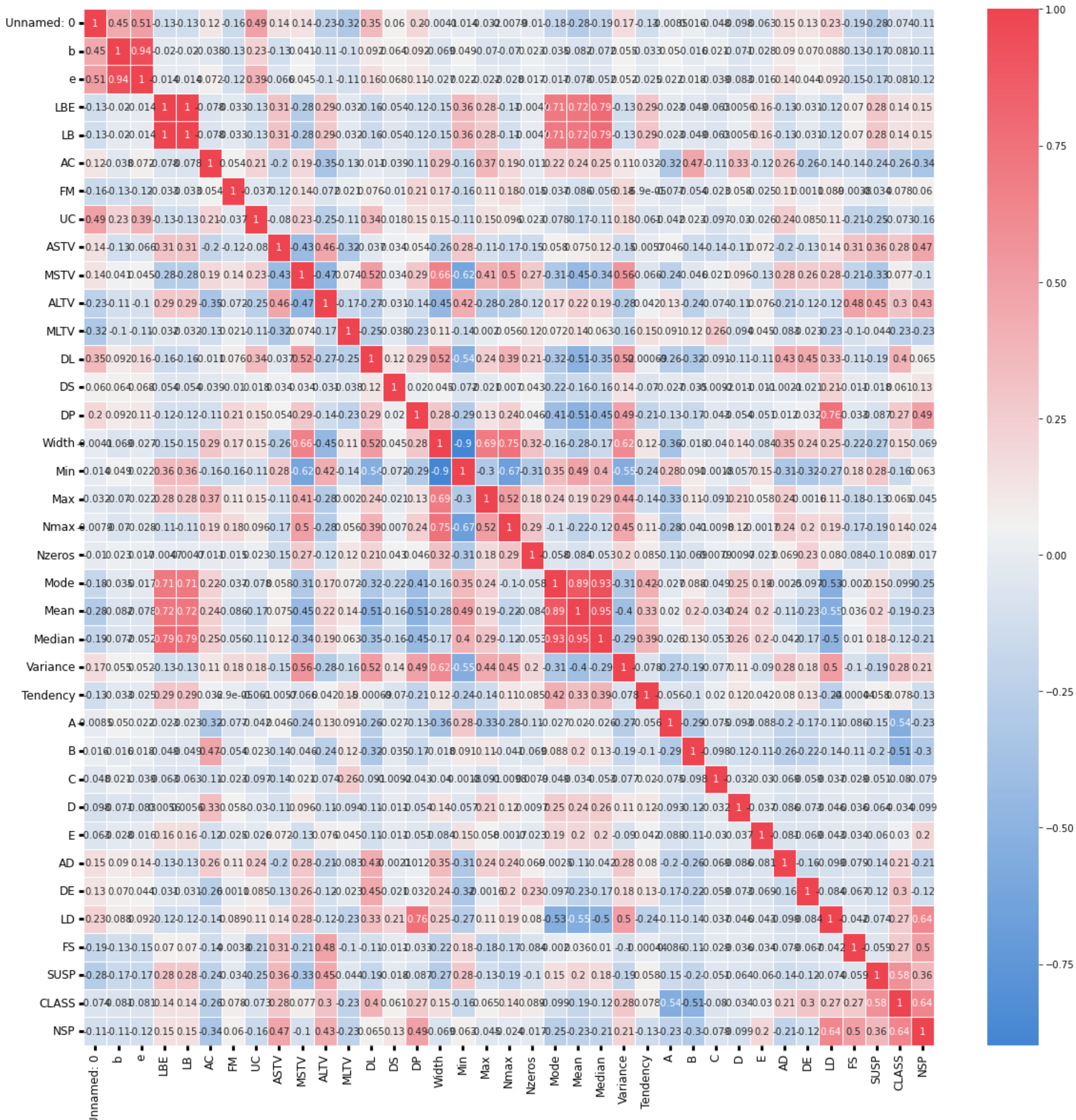
In [5]:

```
# borrowed from: https://www.kaggle.com/code/christopherwsmith/fetal-health-a-quick-guide-to-high-accuracy
def Plotter(plot, x_label, y_label, x_rot=None, y_rot=None,  fontsize=12, fontweight=None, legend=None, save=False,save
    """
    Helper function to make a quick consistent plot with few easy changes for aesthetics.
    Input:
    plot: sns or matplot plotting function
    x_label: x_label as string
    y_label: y_label as string
    x_rot: x-tick rotation, default=None, can be int 0-360
    y_rot: y-tick rotation, default=None, can be int 0-360
    fontsize: size of plot font on axis, defaul=12, can be int/float
    fontweight: Adding character to font, default=None, can be 'bold'
    legend: Choice of including legend, default=None, bool, True:False
    save: Saves image output, default=False, bool
    save_name: Name of output image file as .png. Requires Save to be True.
               default=None, string: 'Insert Name.png'
    Output: A customized plot based on given parameters and an output file

    """
    #Ticks
    ax.tick_params(direction='out', length=5, width=3, colors='k',
               grid_color='k', grid_alpha=1,grid_linewidth=2)
    plt.xticks(fontsize=fontsize, fontweight=fontweight, rotation=x_rot)
    plt.yticks(fontsize=fontsize, fontweight=fontweight, rotation=y_rot)

    #Legend
    if legend==None:
        pass
    elif legend==True:

        plt.legend()
        ax.legend()
        pass
    else:
        ax.legend().remove()

    #Labels
    plt.xlabel(x_label, fontsize=fontsize, fontweight=fontweight, color='k')
    plt.ylabel(y_label, fontsize=fontsize, fontweight=fontweight, color='k')

    #Removing Spines and setting up remianing, preset prior to use.
    ax.spines['top'].set_color(None)
    ax.spines['right'].set_color(None)
    ax.spines['bottom'].set_color('k')
    ax.spines['bottom'].set_linewidth(3)
    ax.spines['left'].set_color('k')
    ax.spines['left'].set_linewidth(3)

    if save==True:
        plt.savefig(save_name)
```

In [6]:

```
fig, ax=plt.subplots(figsize=(20,20))#Required outside of function. This needs to be activated first when plotting in e
cmap = sns.diverging_palette(250, 10, s=80, l=55, n=9, as_cmap=True)
plot=sns.heatmap(df.corr(),annot=True, cmap=cmap, linewidths=1)
Plotter(plot, None, None, 90,legend=False, save=True, save_name='Corr.png')
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when le
gend() is called with no argument.
```



Shades of red are more correlated than blue. Looking at the NSP col/row it appears that LB, DS, DP, ASTV, ALTV, Varaince, SUSP, CLASS have the best correlation.

In [7]:

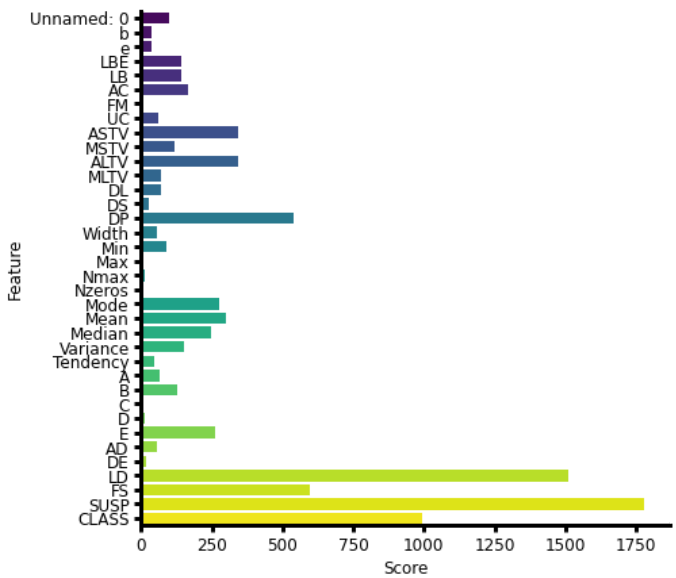
```
# Using KBEst Algo with f_classif to perform ANOVA which:
#determines the degree of linear dependency between the target variable and features.
from sklearn.feature_selection import SelectKBest #Feature Selector
from sklearn.feature_selection import f_classif #ANOVA
```

In [8]:

```
#Feature Selection
X=df.drop(['NSP'], axis=1)
Y=df['NSP']
bestfeatures = SelectKBest(score_func=f_classif, k='all')
fit = bestfeatures.fit(X,Y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(X.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Feature','Score']  #naming the dataframe columns

#Visualize the feature scores
fig, ax=plt.subplots(figsize=(7,7))
plot=sns.barplot(data=featureScores, x='Score', y='Feature', palette='viridis',linewidth=0.5, saturation=2, orient='h')
Plotter(plot, 'Score', 'Feature', legend=False, save=True, save_name='Feature Importance.png')#Plotter function for ae
plot
```

```
No artists with labels found to put in legend.  Note that artists whose label start with an underscore are ignored when le
gend() is called with no argument.
```

Out[8]: <AxesSubplot:xlabel='Score', ylabel='Feature'>



SUSP isnt listed in the offical attribute information, but it was correlated and now shows it has the highet linear dependency. I'm going to exclude it because its not listed and could be an outlier. 250 looks to be a good cut off point for feature selection.

In [41]:

```
#Selection method
selection=featureScores[featureScores['Score']>=200]#Selects features that scored more than 200
selection=list(selection['Feature'])#Generates the features into a list
selection.append('NSP')#Adding the Level string to be used to make new data frame
df_feat=df[selection] #New dataframe with selected features
df_feat = df_feat.drop(columns=['SUSP', 'FS', 'LD', 'E'])
df_feat.head() #Lets take a look at the first 5
```

Out[41]:

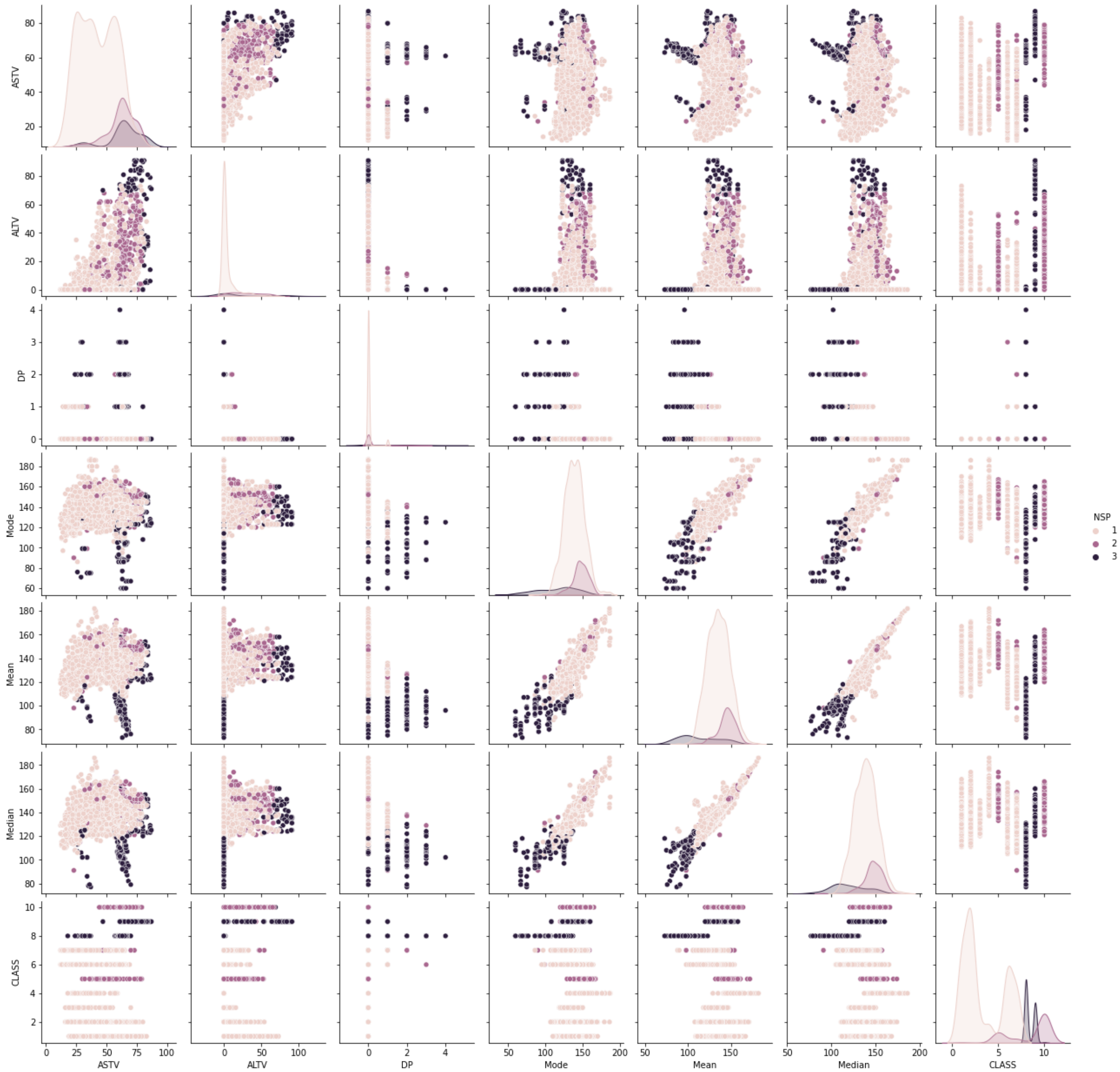
	ASTV	ALTV	DP	Mode	Mean	Median	CLASS	NSP
0	73	43	0	120	137	121	9	2
1	17	0	0	141	136	140	6	1
2	16	0	0	141	135	138	6	1
3	16	0	0	137	134	137	6	1
4	16	0	0	137	136	138	2	1

In [42]:

```python
sns.pairplot(df_feat, hue='NSP')
```

Out[42]:

```
<seaborn.axisgrid.PairGrid at 0x18c53131040>
```



IEEE paper mentions 7 features so this seems to be a good choice!

Classes 2 and 3 are diffcult to distingish here.

Splitting, sclaing, encoding

In [43]:

```python
# make things simple
data = df_feat
```

In [44]:
```
# Encoding the output. labels need to go from 0-2 in order to work with tensor
# 0 = Normal, 1 = Suspect, 2 = Pathologic
# borrowed from https://towardsdatascience.com/pytorch-tabular-multiclass-classification-9f8211a123ab

class2idx = {
    1:0,
    2:1,
    3:2
}

idx2class = {v: k for k, v in class2idx.items()}
data['NSP'].replace(class2idx, inplace=True)
```

In [97]:
```
#Create inputs and targets.
X = data.iloc[:, 0:-1]
y = data.iloc[:, -1]

```

In [98]:
```
print(X.head())
```

```
   ASTV  ALTV  DP  Mode  Mean  Median  CLASS
0    73    43   0   120   137     121      9
1    17     0   0   141   136     140      6
2    16     0   0   141   135     138      6
3    16     0   0   137   134     137      6
4    16     0   0   137   136     138      2
```

In [99]:
```
print(y.head())
```

```
0    1
1    0
2    0
3    0
4    0
Name: NSP, dtype: int64
```

In [100]:
```
# Create split into train, val, test
# Split into train+val and test
# Stratify is being used to have an equal distribution of output classes sets
# test_size is .2, as mentioned in paper

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

In [101]:
```
# Normalize the input. Neural networks need a range of 0,1
# Use MinMaxScaler to transform features
scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# convert inputs and outputs in numpy arrays
X_train, y_train = np.array(X_train),np.array(y_train)
X_test, y_test = np.array(X_test), np.array(y_test)
```

In [102]:
```
X_test.shape
```

Out[102]: (426, 7)

X_val and X_test, .transform was used beacause the validation and test sets should be scaled with the same parameters as the train set to avoid data leakage. fit_transform calculcates scaling values and applies. .transform only applies the calculated values.

Cross-validation will be done in the model building phase

Neural Network

Model parameters

In [103]:

```
# create tensors
X_train = torch.tensor(X_train)
X_test = torch.tensor(X_test)


y_test = torch.tensor(y_test)
y_train = torch.tensor(y_train)
```

In [104]:

```
print(f"Datatypes of training data: X: {X_test.dtype}, y: {y_train.dtype} ")
```

```
Datatypes of training data: X: torch.float64, y: torch.int64
```

In [105]:

```

from sklearn.model_selection import cross_val_score
```

In [106]:

```
#implementing baisc log regression with default as test?
from sklearn import linear_model
from sklearn.linear_model import LogisticRegression

logistic_regression = linear_model.LogisticRegression()
logistic_regression_mod = logistic_regression.fit(X_train, y_train)
print(f"Baseline Logistic Regression: {round(logistic_regression_mod.score(X_test, y_test), 3)}")

pred_logistic_regression = logistic_regression_mod.predict(X_test)
```

```
Baseline Logistic Regression: 0.913
```

Define architecture

Two hidden layers, because of the universal approximation theorm. Input size is 7, output size is 3 classes

In [111]:

```
import torch.nn.functional as F
import torch.nn as nn

class ctgClassifier(nn.Module):
    def __init__(self, dropout=0.5, weight_constraint=1.0):
        super(ctgClassifier, self).__init__()
        self.dropout = nn.Dropout(dropout)

        self.layer_1 = nn.Linear(7, 128)
        self.layer_2 = nn.Linear(128, 64)
        self.layer_out = nn.Linear(64, 3)


    def forward(self, x):
        x = F.relu(self.layer_1(x))
        x = self.dropout(x)
        x = F.relu(self.layer_2(x))
        x = self.dropout(x)
        return x
```

The forward pass of the neural network takes an input tensor x and applies the fully connected layers and activation functions defined. The F.relu() function applies the ctivation function to the output of each fully connected layer. The self.dropout(x) applies dropout regularization to the output of the first and second hidden layers. Finally, the function returns the output tensor x.

In fact, there is a theoretical finding by Lippmann in the 1987 paper “An introduction to computing with neural nets” that shows that an MLP with two hidden layers is sufficient for creating classification regions of any desired shape

Specifically, the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

— Page 198, Deep Learning, 2016

In [112]:

```
p earlying stopping
korch.callbacks import EarlyStopping

stopping = EarlyStopping(monitor='valid_loss', patience = 10, threshold = 0.0001, threshold_mode='rel', lower_is_better=Tru
```

In [113]:

```
#Multi-layer Perceptron classifier.

from skorch import NeuralNetClassifier

net = NeuralNetClassifier(
    ctgClassifier,
    lr=0.1,
    criterion = torch.nn.modules.loss.CrossEntropyLoss,
    optimizer=torch.optim.Adam,
    callbacks=[early_stopping]
)

```

details for tuning: [https://machinelearningmastery.com/how-to-grid-search-hyperparameters-for-pytorch-models/ (https://machinelearningmastery.com/how-to-grid-search-hyperparameters-for-pytorch-models/)](https://machinelearningmastery.com/how-to-grid-search-hyperparameters-for-pytorch-models/)

IEEE paper mentions using cv of 10, so that is what I'll use..

GridSearch

exhaustively searches through all possible combinations of hyperparameters during training the phase. Before we proceed further, we shall cover another cross-validation (CV) methods since tuning hyperparameters via grid search is usually cross-validated to avoid overfitting. Hence, For accelerating the running GridSearchCV we set: n-splits=3, n_jobs=2

In [114]:

```
#Grid Search for the below parameters
from sklearn.model_selection import GridSearchCV
params={
        'module__dropout':[0.5,0.1],
        'module__weight_constraint': [1.0, 2.0, 3.0, 4.0, 5.0],
        'lr':[0.01,0.05,0.1],
        'max_epochs':[50,100],
        'batch_size':[50,100],
        'optimizer__weight_decay':[0.01,0.5]
}
gs=GridSearchCV(net,params,cv=10, scoring=None,n_jobs=-1,verbose=0)
mlp_model=gs.fit(X_train.float(), y_train)
print(gs.best_score_,gs.best_params_)
```

```
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.5646       0.7765        0.7801  0.0502
      2        1.0940       0.8676        0.4157  0.0608
      3        0.8985       0.9265        0.4335  0.0507
      4        0.8068       0.9265        0.3751  0.0512
      5        0.7734       0.9324        0.3520  0.0480
      6        0.7581       0.9294        0.3460  0.0403
      7        0.7562       0.9294        0.3599  0.0358
      8        0.7974       0.9324        0.3516  0.0500
      9        0.7312       0.9294        0.3446  0.0474
     10        0.7751       0.9294        0.3477  0.0364
     11        0.7001       0.9294        0.3489  0.0435
     12        0.7729       0.9294        0.3558  0.0387
     13        0.6475       0.9324        0.3214  0.0470
     14        0.7456       0.9324        0.3501  0.0426
     15        0.6802       0.9324        0.3144  0.0443
     16        0.8304       0.9294        0.3613  0.0374
     17        0.6938       0.9324        0.3162  0.0480
     18        0.6901       0.9324        0.3334  0.0596
     19        0.6848       0.9353        0.3372  0.0544
     20        0.7619       0.9324        0.3481  0.0377
     21        0.7358       0.9324        0.3175  0.0436
     22        0.7513       0.9294        0.3473  0.0437
     23        0.7001       0.9324        0.3170  0.0498
     24        0.7885       0.9324        0.3658  0.0413
     25        0.7584       0.9324        0.3137  0.0430
     26        0.7416       0.9324        0.3274  0.0524
     27        0.7244       0.9324        0.3183  0.0488
     28        0.7660       0.9324        0.3634  0.0508
     29        0.6728       0.9324        0.3169  0.0359
     30        0.7246       0.9324        0.3582  0.0369
     31        0.7419       0.9324        0.3399  0.0544
     32        0.6886       0.9294        0.3136  0.0362
     33        0.7114       0.9324        0.3373  0.0377
     34        0.7221       0.9294        0.3247  0.0411
     35        0.7524       0.9324        0.3215  0.0383
     36        0.7605       0.9324        0.3423  0.0411
     37        0.7222       0.9324        0.3144  0.0362
     38        0.7911       0.9294        0.3402  0.0456
     39        0.6681       0.9353        0.3321  0.0410
     40        0.7178       0.9324        0.3325  0.0396
     41        0.6439       0.9294        0.3150  0.0402
Stopping since valid_loss has not improved in the last 10 epochs.
0.9288235294117648 {'batch_size': 100, 'lr': 0.01, 'max_epochs': 100, 'module__dropout': 0.1, 'module__weight_constraint': 2.0, 'optimizer__weight_decay': 0.01}
```

4.9 results: Severally overfitting

Stopping since valid_loss has not improved in the last 10 epochs. 0.9888888888888889 {'batch_size': 50, 'lr': 0.05, 'max_epochs': 100, 'module__dropout': 0.1, 'module__weight_constraint': 3.0, 'optimizer__weight_decay': 0.01}Stopping since valid_loss has not improved in the last 10 epochs.

Continue working through fetal class kaggle. almost have it figured out!. [https://discuss.pytorch.org/t/runtimeerror-mat1-and-mat2-must-have-the-same-dtype/166759 (https://discuss.pytorch.org/t/runtimeerror-mat1-and-mat2-must-have-the-same-dtype/166759)](https://discuss.pytorch.org/t/runtimeerror-mat1-and-mat2-must-have-the-same-dtype/166759) pytorch_most_common_errors

In [115]:

```
print(gs.best_score_,gs.best_params_)
```

```
0.9288235294117648 {'batch_size': 100, 'lr': 0.01, 'max_epochs': 100, 'module__dropout': 0.1, 'module__weight_constraint': 2.0, 'optimizer__weight_decay': 0.01}
```

In [ ]:
```
#mlp_model = gs.fit(X_train.float(), y_train)
```

Prediciton

In [116]:
```
from sklearn.metrics import accuracy_score
```

In [117]:
```
#train
y_pred_train = mlp_model.predict(X_train.float())
#test
y_pred_test = mlp_model.predict(X_test.float())
```

Is the next step necessary?

In [118]:
```
#getting the recall_score on train
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score

print("recall_score:",recall_score(y_train, y_pred_train,average='macro'))
print("accuracy_score:",accuracy_score(y_train, y_pred_train))
from sklearn.metrics import classification_report,confusion_matrix
confusion_matrix(y_train, y_pred_train)
```

```
recall_score: 0.8105692357748753
accuracy_score: 0.9294117647058824
```

Out[118]:
```
array([[1300,   12,   11],
       [  46,  188,    2],
       [   6,   43,   92]], dtype=int64)
```

getting the recall_score on test

In [119]:
```
#getting the recall_score on test
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score

print("recall_score:",recall_score(y_test, y_pred_test,average='macro'))
print("accuracy_score:",accuracy_score(y_test, y_pred_test))
from sklearn.metrics import classification_report,confusion_matrix
confusion_matrix(y_test, y_pred_test)
```

```
recall_score: 0.7352543345294009
accuracy_score: 0.9061032863849765
```

Out[119]:
```
array([[327,   3,   2],
       [ 18,  40,   1],
       [  1,  15,  19]], dtype=int64)
```

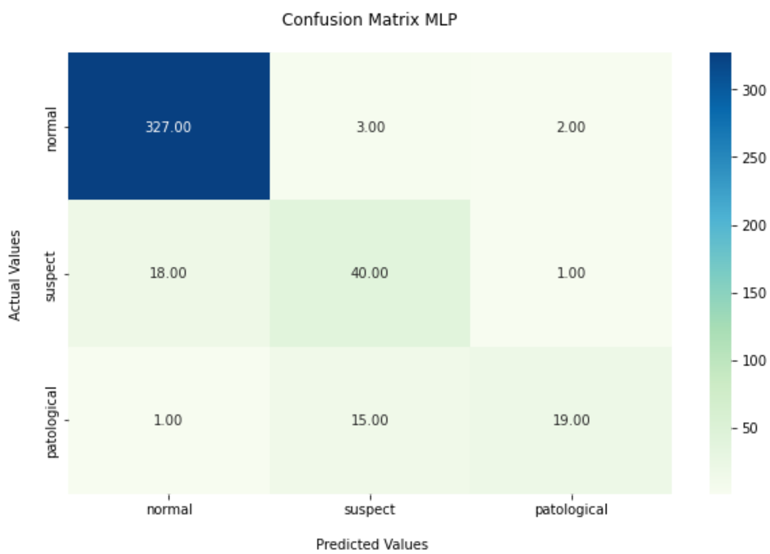
In [120]:
```
#Binarize labels in a one-vs-all fashion to use the Y_test_Roc values could be used while plotting the ROC curves
from sklearn.preprocessing import label_binarize
Y_test_ROC = label_binarize(y_test, classes=[0, 1, 2])
print(Y_test_ROC)
```

```
[[1 0 0]
 [1 0 0]
 [1 0 0]
 ...
 [1 0 0]
 [0 1 0]
 [1 0 0]]
```

In [121]:
```
report = classification_report(y_test, y_pred_test)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       332
           1       0.69      0.68      0.68        59
           2       0.86      0.54      0.67        35

    accuracy                           0.91       426
   macro avg       0.83      0.74      0.77       426
weighted avg       0.90      0.91      0.90       426
```

In [122]:
```
import seaborn as sns

plt.figure(figsize=(10,6))
fx=sns.heatmap(confusion_matrix(y_test,y_pred_test), annot=True, fmt=".2f",cmap="GnBu")
fx.set_title('Confusion Matrix MLP\n');
fx.set_xlabel('\n Predicted Values\n')
fx.set_ylabel('Actual Values\n');
fx.xaxis.set_ticklabels(['normal','suspect','patological'])
fx.yaxis.set_ticklabels(['normal','suspect','patological'])
plt.show()
```



In [123]:
```
from sklearn.preprocessing import LabelBinarizer

label_binarizer = LabelBinarizer().fit(y_train)
y_onehot_test = label_binarizer.transform(y_test)
y_onehot_test.shape  # (n_samples, n_classes)
```

Out[123]: (426, 3)

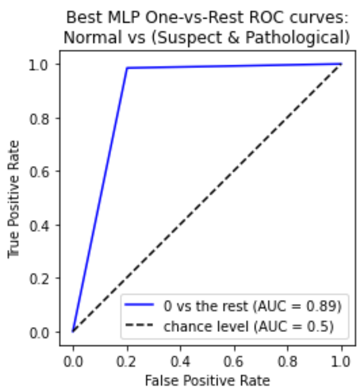
In [124]:
```
class_of_interest = 0
class_id = np.flatnonzero(label_binarizer.classes_ == class_of_interest)[0]
class_id
```

Out[124]: 0

In [125]:
```
predBi = label_binarizer.transform(y_pred_test)
```

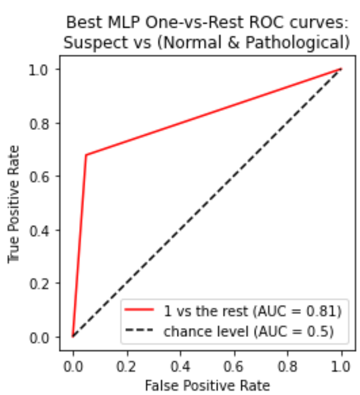
In [126]:

```
import matplotlib.pyplot as plt
from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from_predictions(
    y_onehot_test[:, class_id],
    predBi[:, class_id],
    name=f"{class_of_interest} vs the rest",
    color="blue",
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Best MLP One-vs-Rest ROC curves:\nNormal vs (Suspect & Pathological)")
plt.legend()
plt.show()
```

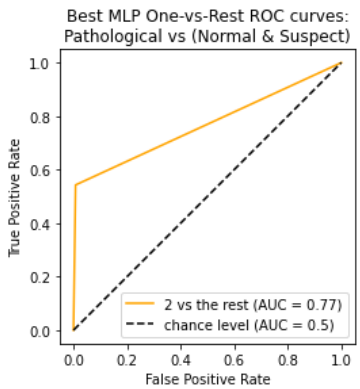


In [127]:

```
class_of_interest = 1
class_id = np.flatnonzero(label_binarizer.classes_ == class_of_interest)[0]
class_id

RocCurveDisplay.from_predictions(
    y_onehot_test[:, class_id],
    predBi[:, class_id],
    name=f"{class_of_interest} vs the rest",
    color="red",
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Best MLP One-vs-Rest ROC curves:\nSuspect vs (Normal & Pathological)")
plt.legend()
plt.show()
```



In [128]:
```
class_of_interest = 2
class_id = np.flatnonzero(label_binarizer.classes_ == class_of_interest)[0]
class_id

RocCurveDisplay.from_predictions(
    y_onehot_test[:, class_id],
    predBi[:, class_id],
    name=f"{class_of_interest} vs the rest",
    color="orange",
)
plt.plot([0, 1], [0, 1], "k--", label="chance level (AUC = 0.5)")
plt.axis("square")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Best MLP One-vs-Rest ROC curves:\nPathological vs (Normal & Suspect)")
plt.legend()
plt.show()
```



Classification Report: Report which includes Precision, Recall and F1-Score. Precision - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations.

Precision = TP/TP+FP

Recall (Sensitivity) - Recall is the ratio of correctly predicted positive observations to the all observations in actual class - yes.

Recall = TP/TP+FN

F1 score - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it’s better to look at both Precision and Recall.

SVM

In [129]:
```
from sklearn import svm
```

In [130]:
```
svmclassifier=svm.SVC(kernel='rbf')
svmclassifier.fit(X_train, y_train)
svmpred=svmclassifier.predict(X_test)
```

In [131]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

#param range
param_grid = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf','linear','poly'],
              'degree' : [2, 3, 4]
             }
grid = GridSearchCV(SVC(), param_grid,scoring='recall_macro', refit = True,cv=10, verbose = 0)

# fitting the model for grid search
svmModel = grid.fit(X_train,y_train)
```

In [132]:

```
# print best parameter after tuning
print(grid.best_params_)

# print how our model looks after hyper-parameter tuning
print(grid.best_estimator_)
```

```
{'C': 1000, 'degree': 2, 'gamma': 1, 'kernel': 'rbf'}
SVC(C=1000, degree=2, gamma=1)
```

old best params: {'C': 100, 'degree': 2, 'gamma': 0.1, 'kernel': 'rbf'} SVC(C=100, degree=2, gamma=0.1)

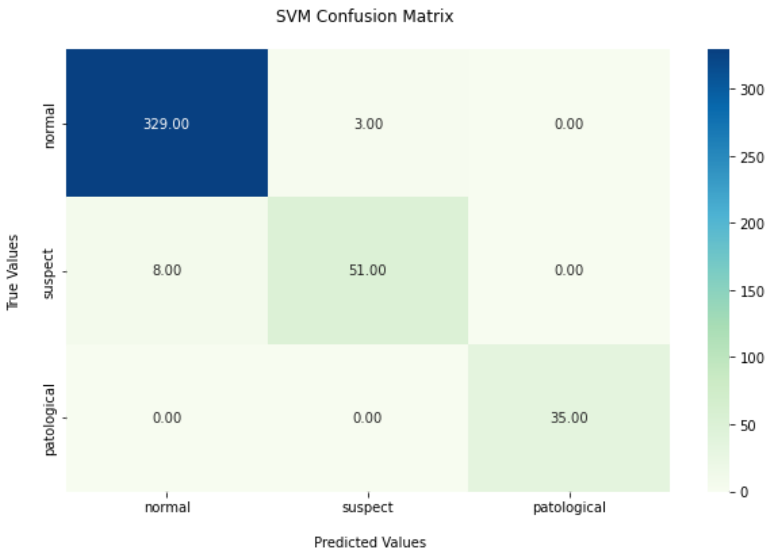
In [133]:

```
grid_predictions = grid.predict(X_test)


# print classification report
print(classification_report(y_test, grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       332
           1       0.94      0.86      0.90        59
           2       1.00      1.00      1.00        35

    accuracy                           0.97       426
   macro avg       0.97      0.95      0.96       426
weighted avg       0.97      0.97      0.97       426
```

In [134]:

```
#show confuse

plt.figure(figsize=(10,6))
fx=sns.heatmap(confusion_matrix(y_test,grid_predictions), annot=True, fmt=".2f",cmap="GnBu")
fx.set_title('SVM Confusion Matrix \n');
fx.set_xlabel('\n Predicted Values\n')
fx.set_ylabel('True Values\n');
fx.xaxis.set_ticklabels(['normal','suspect','patological'])
fx.yaxis.set_ticklabels(['normal','suspect','patological'])
plt.show()
```



In [135]:

```
normalTN_SVM = ((51+0+0+35)/426)*100
normalTN_MLP = ((36+0+15+17)/426)*100


suspectTN_SVM =((331+0+0+35)/426)*100
suspectTN_MLP = ((329+2+3+17)/426)*100

pathTN_SVM = ((331+1+51+8)/426)*100
pathTN_MLP = ((329+1+23+36)/426)*100


print("Normal TN SVM:", normalTN_SVM)
print("Normal TN MLP:", normalTN_MLP)

print("Suspect TN SVM:", suspectTN_SVM)
print("Suspect TN MLP:", suspectTN_MLP)

print("Pathlogic TN SVM:", pathTN_SVM)
print("Pathlogic TN MLP:", pathTN_MLP)
```

```
Normal TN SVM: 20.187793427230048
Normal TN MLP: 15.96244131455399
Suspect TN SVM: 85.91549295774648
Suspect TN MLP: 82.3943661971831
Pathlogic TN SVM: 91.78403755868545
Pathlogic TN MLP: 91.31455399061032
```

In [136]:

```
print("recall_score:",recall_score(y_test, grid_predictions, average='macro'))
print("accuracy_score:",accuracy_score(y_test, grid_predictions))
```

```
recall_score: 0.9517902116942345
accuracy_score: 0.9741784037558685
```

Export best model

In [137]:

```
bestMLP=mlp_model
import joblib
```

In [138]:

```
filename = 'best_MLP_model.joblib'
joblib.dump(bestMLP, filename)
```

Out[138]:

```
['best_MLP_model.joblib']
```

In [139]:

```
# load the model from disk
svmload=joblib.load('best_MLP_model.joblib')
```

In [ ]:

```

```