Classification Significance of Neural Networks on rs-fMRI data

Varaprasad Rao Kurra
vkurra1@student.gsu.edu

Sravanthi Malepati
smalepati1@student.gsu.edu

***Abstract*—The aim is to do a systematic study on the neural networks and find an optimal neural network that will give us the best accuracy in classifying the ADHD rs-fMRI data. We obtain the dataset from the Nilearn learn package. Nilearn is a Python module for fast and easy statistical learning on neuroimaging data. Our work involves practical steps of using machine learning to analyze rs-fMRI data and, more specifically to discriminate Attention Deficit Hyperactivity Disorder (ADHD) from healthy controls. We will discuss (1) feature extraction using masks (2) The benefits and drawbacks of recurrent neural networks (RNN) and particularly long short-term memory networks (LSTM) for classifying fMRI data (3) hypothesis testing and its use in model evaluation.**

Keywords – Neural Networks, LSTM, ADHD, masking, ICA.

Introduction

Traditional Machine Learning models fail to understand the patterns in the fMRI data. We need machine learning models that will be able to learn from the data. Identify the patterns so that the accuracy of the model can be increased during the training phase itself. What is good about the neural networks is that we have two phases to learn from the data: forward propagation and backward propagation. The neural networks learn and adjust the neural network's weights and help obtain the results with reasonable accuracy. Our experiment involves the Data Preparation state, where we mask our data to eliminate the unwanted features. Next, we train our neural networks and compare the models' accuracy to know which model is the best fit for the fMRI data.

I. Data Preparation

A. Feature Extraction

FMRI images are 4D matrices reflecting the activation level of each voxel at the three-dimensional space and time. However, often, the relevant information is portrayed by a subset of this data. For example, here- we are only curious about the resting-state networks. To omit the irrelevant data, we apply masks. Masks are simply filters that pass the desired subset of the data while dropping the rest. Practically, masks replace the activation value of unwanted voxels to 0.

There are many styles to mask fMRI data, which is mostly determined by the analysis's goal. This tutorial focuses on classifying ADHD patients from controls via their resting-state networks. Thus, we apply Smith's rs-fMRI components atlas (Smith et al.,2009). Smith atlas reflects seventy resting-state networks (RSN) acquired using an independent component analysis (ICA) on thousands of healthy patients. We prefer Smith atlas over the use of dataset-specific ICA components because it helps avoid double-dipping (i.e., using the data twice), leading to overfitting.

B. Feature Extraction Implementation

The template is used to format the paper and style the text. All margins, column widths, line spaces, and text fonts are prescribed; please do not alter them. You may note peculiarities. For example, the head margin in this template measures proportionately more than is customary. This measurement and others are deliberate, using specifications that anticipate your paper as one part of the entire proceedings and not as an independent document. Please do not revise any of the current designations.

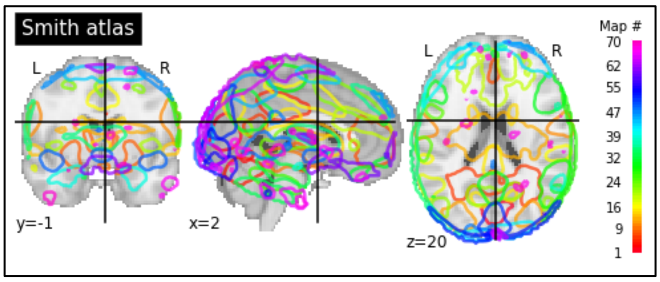
The Smith's atlas is available via Nilearn dataset:

```
from nilearn import plotting
from nilearn import datasets



## load the smith (ICA based) mask
smith_atlas = datasets.fetch_atlas_smith_2009()
smith_atlas_rs_networks = smith_atlas.rsn70

## plot
plotting.plot_prob_atlas(smith_atlas_rs_networks,
                         title='Smith atlas',
                         colorbar=True)
plotting.show()
```

The code snippet gives us a resting state network of 70 regions. We have three options in choosing the number of networks. Firstly, we can select 10 regions at a time. Secondly, we can select the top 20 regions and 70 regions of the resting state network. To increase the accuracy and get a good picture of the data, we select 70 rs networks as a masker to drop the unnecessary features.

In the below picture, we can see the three views of the image that we will be using to extract the features from the given fMRI scan. We can see the map next to the images that provide a picture of the number of resting-state networks. The ADHD dataset is also available on Nilearn. We see that the image contains 73*61*61 voxels over 176 timestamps from the first image's header.

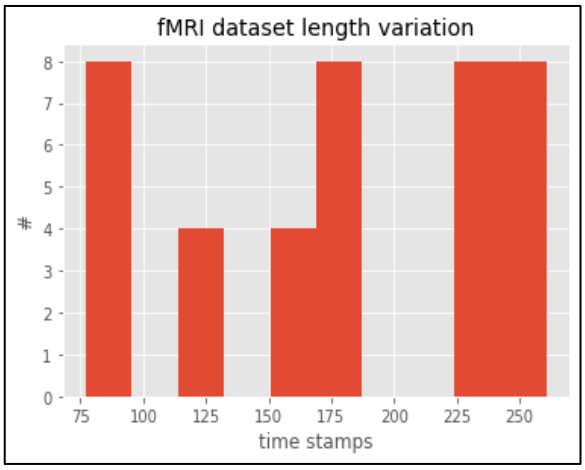
Additionally, each voxel is about 3mm³. It is important to note that this information may not be uniform across the whole dataset.



Why masking. Applying a standardization could contribute to the features' robustness. In masking, it can help enhance the signal by centering and normalizing the slices for each time-series. Considering the data confounds as part of the transformation process also enhances the signal by removing confounding noise.

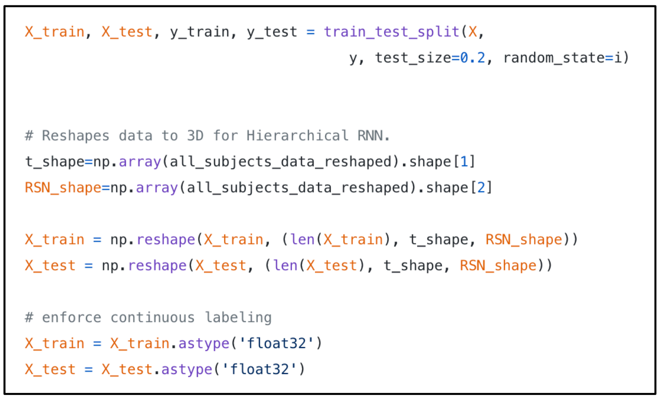
C. *Insights of the Data*

As mentioned before, a dataset might not hold a homogenous scanning length — these 40 subjects present quite a considerable variation. However, most machine learning algorithms (Keras included) require a uniform shape across all subjects. To optimize for keeping data, we can use padding; append each subject with zeros after the end of its scan to match the longest scan's length. In addition to padding, we reshape the data to fit Keras's requirements; (40,261,10) implies that we have 40 subjects with 261 timestamps long sample over 10 regions.



D. *Data Preparation*

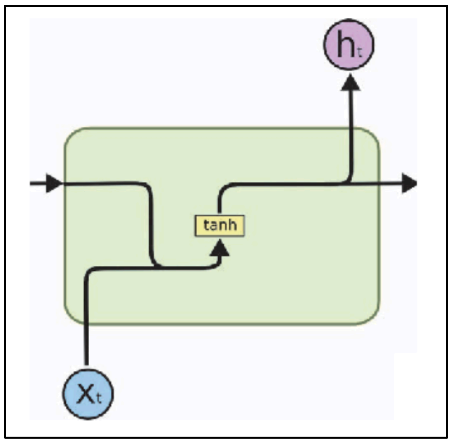
We use the train/test split paradigm to make sure the model is tested on entirely new data. The function below randomly splits the data into train and test and reshapes each part according to the model's requirements. We will see the code snippet about how we performed the tarin test split in the upcoming steps. How did we reshape the data?



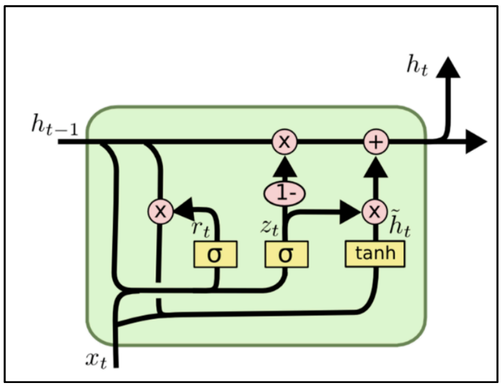
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y, test_size=0.2, random_state=i)


# Reshapes data to 3D for Hierarchical RNN.
t_shape=np.array(all_subjects_data_reshaped).shape[1]
RSN_shape=np.array(all_subjects_data_reshaped).shape[2]

X_train = np.reshape(X_train, (len(X_train), t_shape, RSN_shape))
X_test = np.reshape(X_test, (len(X_test), t_shape, RSN_shape))

# enforce continuous labeling
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

II. Neural Network Models

Simple RNN: Here, there is a simple multiplication of Input (Xt) and Previous Output (ht-1). Passed through Tanh activation function. No Gates present.



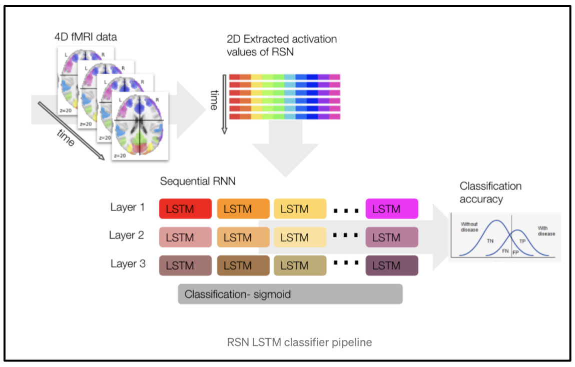
Gated Recurrent Unit (GRU): Here, an **Update gate** is introduced to decide whether to pass Previous O/P (ht-1)to the next Cell (as ht) or not. Forget gate is nothing but additional Mathematical Operations with a new set of Weights (Wt).



Long Short Term Memory Unit (LSTM):- Here, 2 more Gates are introduced (Forget and Output) in addition to the Update gate of GRU. And again, as above, these are additional Mathematical Operations on the same inputs (Xt and ht-1). So overall, LSTM has introduced 2 Math operations having 2 new sets of Weights.

Long short-term memory (LSTM) models also provide some benefits in learning fMRI data. The main reason is that, unlike most machine learning or deep learning methods, they manage to keep the contextual information of the inputs — thus incorporate details from previous parts of the input sequence while processing a current one. That said, being highly contextual isn't always such a good thing. There are cases where LSTMs are not the best choice; being contextual could lead to an over-interpretation of the data. Besides, LSTMs could take longer to run than a simple NN, and they might have far more parameters to tune.

fMRI data represents dynamic brain activity over time, thus using LSTMs enables taking advantage of the temporal information (that otherwise would have been lost) when analyzing functional connectivity.



The above pipeline shows our step-by-step approach to find our experiment. Firstly, we get the raw ready to go data from the nilearn defined package. Next, we will transform these 4D matrices into meaningful data, i.e., 2D extracted RSN values.

A common enhancement for LSTM is convolutional neural networks (CNN), which supports analyzing the spatial structure. However, here, we extracted seventy discrete values reflecting independent components of the activation of whole networks — and thus gave up on the spatial property of the data. Therefore, CNN would be unlikely to be useful.

An issue with LSTMs is that they can easily overfit training data, reducing their predictive capacity. A solution to this problem is via regularization, which hinders the model's overfitting tendency. A conventional regularization in LSTM networks is the dropout, which probabilistically excludes units from the layer connection. There are two types of dropouts- input dropout and recurrent dropout. A dropout on the input means that for a given probability, the data on the input connection to each LSTM unit will be excluded from node activation and weight updates (see the *dropout* argument). The dropout on the recurrent input acts the same way but on the recurrent connection (see the *reccurent_dropout* argument). However, it is important not to over-regularize as it will hinder the model from learning (which could be detected by strict non-indicative prediction).

The model we present here is a sequential model with three stacked LSTM layers and one dense layer with sigmoid activation. To be frank, there is no one right answer to how one chooses their hyperparameters. There is a lot of trial and error and rules of thumb. We present a few that we have collected.

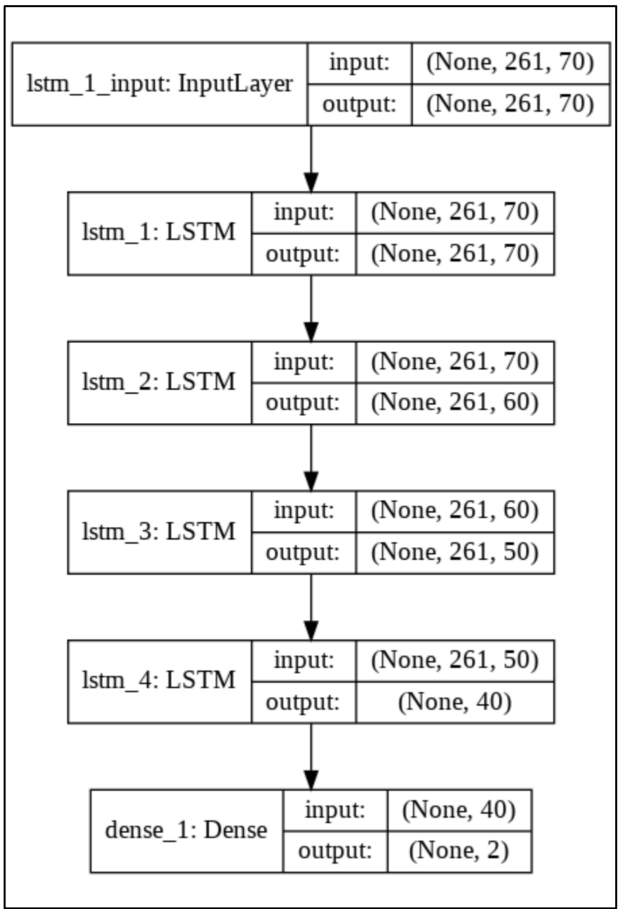
- Generally, we would like to have at least two **hidden layers** (not including the last layer) since the power of NNs stems from their depth (i.e., a zero-layer can only represent linear functions.).
- We would like to start with a **number of units** smaller or equal to the input size and decrease it (by about a half at the time) until reaching the final layer.
- In the case of classification, the **number of units at the output layer** should be equal to the number of categories. Typically, a binary classification problem has one output.
- The **output layer activation** is mostly sigmoid for binary classification, SoftMax for a multi-class classifier and linear for regression.
- The **classification loss function** should be matched to the number of labels and their type.
- The accuracy **metric** is excellent to show the percent of correct classifications. You can use more than one!
- The more **epochs**, the better; start with 30 and follow your **validation set** to decrease loss and improve accuracy. If you see improvement, increase the number of epochs; otherwise, go back to the drawing board.
- Choosing the **optimizer** based on your knowledge of your data's properties and the depth of the network.

LSTM Model

The code snippet to build the LSTM model is as follows:

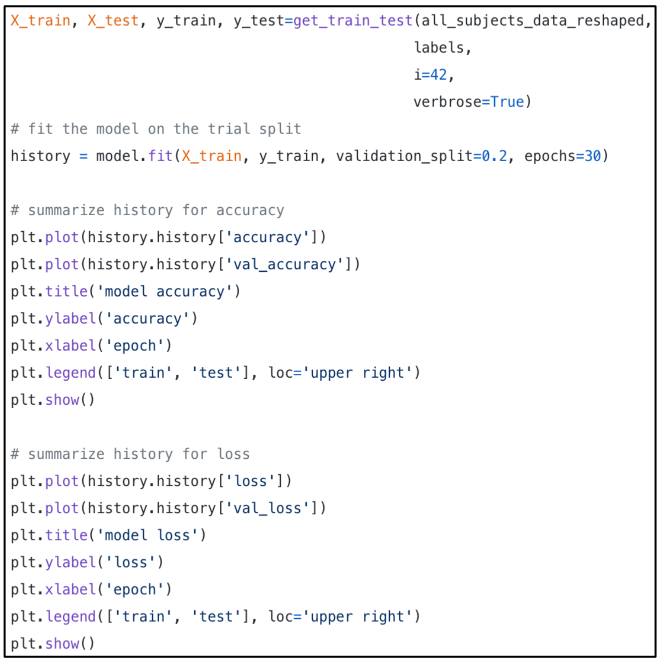
```
model = Sequential()

# LSTM layers -
# Long Short-Term Memory layer - Hochreiter 1997.
t_shape=np.array(all_subjects_data_reshaped).shape[1]
RSN_shape=np.array(all_subjects_data_reshaped).shape[2]

model.add(LSTM(units=70, # dimensionality of the output space
               dropout=0.4, # Fraction of the units to drop (inputs)
               recurrent_dropout=0.15, # Fraction of the units to drop (recurent state)
               return_sequences=True, # return the last state in addition to the output
               input_shape=(t_shape,RSN_shape)))

model.add(LSTM(units=60,
               dropout=0.4,
               recurrent_dropout=0.15,
               return_sequences=True))

model.add(LSTM(units=50,
               dropout=0.4,
               recurrent_dropout=0.15,
               return_sequences=True))

model.add(LSTM(units=40,
               dropout=0.4,
               recurrent_dropout=0.15,
               return_sequences=False))


model.add(Dense(units=2,
                activation="sigmoid"))
```

LSTM model which we have built looks as shown in the below screenshot.



III. Training the Model

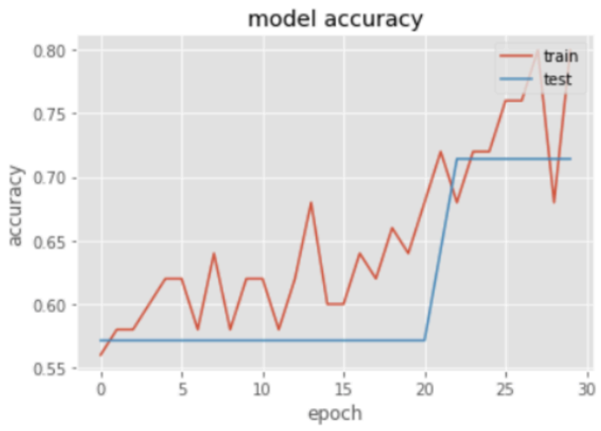
After building the model, we train the model and see if it manages to learn. To train the model, we split the dataset by using epochs.

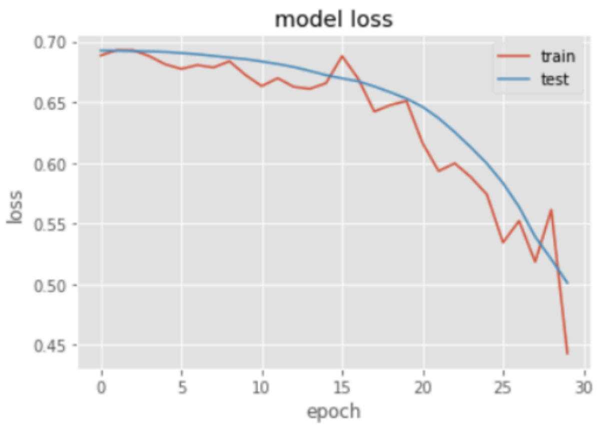


```
X_train, X_test, y_train, y_test=get_train_test(all_subjects_data_reshaped,
                                               labels,
                                               i=42,
                                               verbrose=True)
# fit the model on the trial split
history = model.fit(X_train, y_train, validation_split=0.2, epochs=30)

# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

Here, one epoch is when an entire dataset is passed forward and backward through the neural network model only once. Since one epoch is too big to feed to the computer at once, we divide it into several batches. Accuracy is the number of correctly predicted data points out of all the data points, and loss indicates how bad the model's prediction is. If the model's prediction is perfect, the loss is zero; otherwise, the loss is more significant.

The above code snippet is to train the model on the split dataset, and the graph shows the results of accuracy and loss. From the graph results, we see some increase in the model's accuracy and some decrease in the model's loss as the number of epochs increases —additionally, both the training and validation sets showing similar trends. Thus, we are (1) having a working model and (2) probably not overfitting.





Hypothesis Testing

We use a hypothesis testing framework to evaluate the model since it helps obtain both the model's accuracy and its significance. To calculate the model's significance, we use bootstrapping. Bootstrapping is a computer-based method for statistical inference without relying on too many assumptions. It helps us approximate the properties of the population by estimates from small data samples.

Our null hypothesis, to be tested, states that there are no differences between ADHD patients to controls; thus, the accuracy of comparing the two using our model should be about 0.5.

To evaluate this hypothesis, we will repeatedly resample with replacement from the data, splitting it into a train and test parts, fitting the model, predicting the labels of test data, and calculating the accuracy. Such repetition will create a distribution of accuracies. Such resampling that follows the central limit theorem is likely to approach a Gaussian shape the larger the number of iterations we use. Thus, we could employ statistical tests that require normal distribution, like t-test.

If 0.5 is placed at the distribution tails, we can conclude that it is very unlikely that the distribution is centered around 0.5 and reject the null hypothesis. Otherwise, we would fail to reject it. The next code snippet runs the bootstrapped experiment.

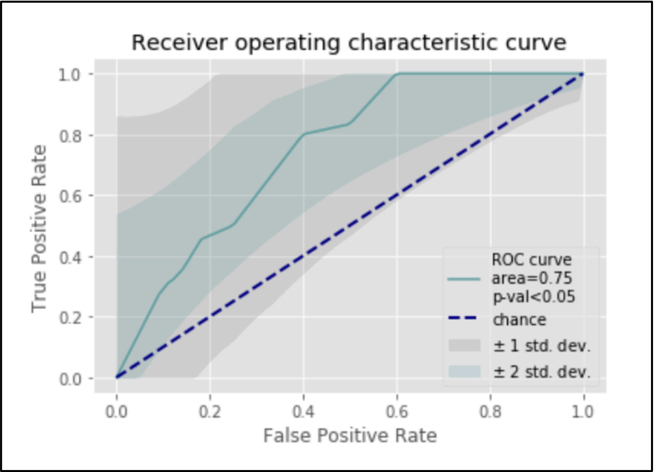
```
def boostrapping_hypothesis_testing(X_train, y_train, X_test, y_test,
                                    n_iterations=100, n_epochs=50):

  '''
  hypothesis testing function
  X_train, y_train, X_test, y_test- the data
  n_iterations- number of bootdtaping iterations
  n_epochs - number of epochs for model's training
  '''

  accuracy=[] ## model accuracy
  roc_msrmnts_fpr=[] ## false positive rate
  roc_msrmnts_tpr=[] ## true positive rate

  # run bootstrap
  for i in range(n_iterations):
    # prepare train and test sets
    X_train, X_test, y_train, y_test=get_train_test(all_subjects_data_reshaped,
                                          labels, i=i, verbrose=False)
    # fit model
    print('fitting..')
    model.fit(X_train, y_train, epochs=n_epochs)

    # evaluate model
    print('evaluating..')
    y_pred=model.predict(X_test)
    y_test_1d=[i[0] for i in y_test]
    y_pred_1d=[1.0 if i[0]>.5 else 0.0 for i in y_pred]

    fpr, tpr, _ = roc_curve(y_test_1d, y_pred_1d)

    acc_score = accuracy_score(y_test_1d, y_pred_1d)

    accuracy.append(acc_score)
    roc_msrmnts_fpr.append(fpr)
    roc_msrmnts_tpr.append(tpr)

    return accuracy, roc_msrmnts_fpr, roc_msrmnts_tpr
```

Lastly, we presented the results using the Receiver operating characteristic (ROC) curve. The ROC curve reflects the model's sensitivity and specificity by plotting the true-positive rate against the false-positive rate. We chose to present the ROC curve's median value since it presents a more robust measurement in case the data is skewed.

Here, Sensitivity is a measure of the proportion of actual positive cases that got predicted as positive, and it is also called Recall. Specificity is a measure of the proportion of people not suffering from the disease who got predicted correctly as those who are not suffering from the disease. The below code snippet plots the ROC curve for the bootstrapped results calculated above.

```
def plot_roc_curve(fpr_vals, tpr_vals, roc_auc, p_val):
  '''
  This function plots the median value of the roc for the boostrapped
    results calculated above.

  fpr stand for false-positive rate
  tpr stands for true-positive rate
  roc_auc is the area under curve
  '''

  ## get the values
  N=len(fpr_vals)
  tprs=[]
  median_fpr=np.linspace(0, 1, 100)
  tprs=[interp(median_fpr, fpr_vals[i], tpr_vals[i]) for i in range(N)]
  std_tpr = np.std(tprs, axis=0)

  mean_tpr = np.mean(tprs, axis=0)
  median_tpr=np.median(tprs, axis=0)
  median_tpr[-1] = 1.0

  tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
  tprs_lower = np.maximum(mean_tpr - std_tpr, 0)

  median_auc_roc=np.median(roc_auc)

  ## plot
  plt.plot(median_fpr, median_tpr, color='cadetblue',
           label='ROC curve \narea={} \np-val~={}'.\
            format(np.round(median_auc_roc,2),
                   np.round(p_val,5)))
  plt.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2,
                 label=r'$\pm$ 1 std. dev.')

  plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label=r'chance')

  plt.xlabel('False Positive Rate')
  plt.ylabel('True Positive Rate')
  plt.title('Receiver operating characteristic curve')
  plt.legend(loc="lower right")
```

The below ROC curve shows a significant classification. We see that the median value and one SD around it is entirely placed above the chance diagonal. The 95% confidence intervals presented by 2 SD around the ROC curve expands mostly at the bottom left point. This concludes that the model is likely to express higher sensitivity but lower specificity.



The model's sensitivity is the proportion of patients identified correctly to have the disease upon the total number of patients with the disease. The model's specificity describes the proportion of patients identified correctly to not have the disease upon the total number of patients who do not have the disease. Usually, these two display an inverse relation. However, it is common in diagnostic experiments to favor sensitivity- thus not missing any ill patients, and such a decision is highly related to the nature of the hypothesis.

IV. Conclusion

We have built an LSTM model to classify ADHD patients from healthy controls via rs-fMRI. We have discussed hypothesis testing and demonstrated its benefits in a diagnostic experiment. Our Future work includes comparing the neural work models in classifying the ADHD rs-fMRI data.

References

[1] Borlase,N., Melzer, T.R., Eggleston, M.J., Darling, K.A., & Rucklidge, J.J, "Resting-state networks and neurometabolites in children with ADHD after 10 weeks of treatment with micronutrients: results of a randomised placebo-controlled trial, 2019.

[2] Dvornek, N.C., Ventola, P., Pelphery, K.A., & Duncan, J.S.(2017,September). Identifying austism from resting state fMRI using long short-term memory networks. In International Workshop on Machine Learning in Medical Imaging(pp. 362-370) . Springer, Cham.

[3] Parikh, R., Mathai, A., Parikh, S., Sekhar, G. C., & Thomas, R. (2008). Understanding and using sensitivity, specificity and predictive values. Indian journal of ophthalmology.

[4] https://nilearn.github.io/modules/generated/nilearn.datasets.fetch_adhd.html